(WS-430) AT91RM9200 评估板

用户手册

版本号：3.0

2005 年 10 月

北京微芯力科技有限公司

http://www.winsilicon.com

目 录

上部 Linux 程序开发基础篇 3
第一章 硬件准备 3
第二章 运行 Linux 操作系统下的例程 4
2．1 启动Uboot并加载Linux 4
2．2 通过ftp方式运行testled例程 11
2．3 加载卸载优盘实验 14
2．4 优盘读写测试实验 17
第三章 交叉编译环境和调试环境的建立 18
3．1 建立交叉编译环境 18
3．2 Linux内核编译 18
下部 Linux 程序开发提高篇 20
第四章 Linux 驱动、应用程序的编写 20
4．1 编写Linux应用程序 20
4．2 调试Linux驱动程序 21
第五章 Linux 下设备驱动程序的开发 25
5．1 基于设备接口的Linux驱动程序 25
5．2 基于模块化的字符设备驱动框架 27
第六章 如何创建文件系统 34
附录
附录一 Linux 常用命令 35
附录二 超级终端的使用和配置 39
附录三 MINICOM 的使用和配置 42

上部 Linux 程序开发基础篇

第一章 硬件准备

根据开发板配套的部件，首先做好硬件的连接。
<1> AT91RM9200 开发板一套
<2> PC 机一台（Windows98/Windows2000/WindowsXP and RedHat Linux8.0 及以上）
<3> DB9 串口线一根（交叉）
<4> Ethernet 交叉网线一根
<5> 5V 直流稳压电源一只
硬件连接如图 1 所示：

注意：图中的位置仅作参考，这里用到的是调试串口 J1，而不是标准串口 J3！

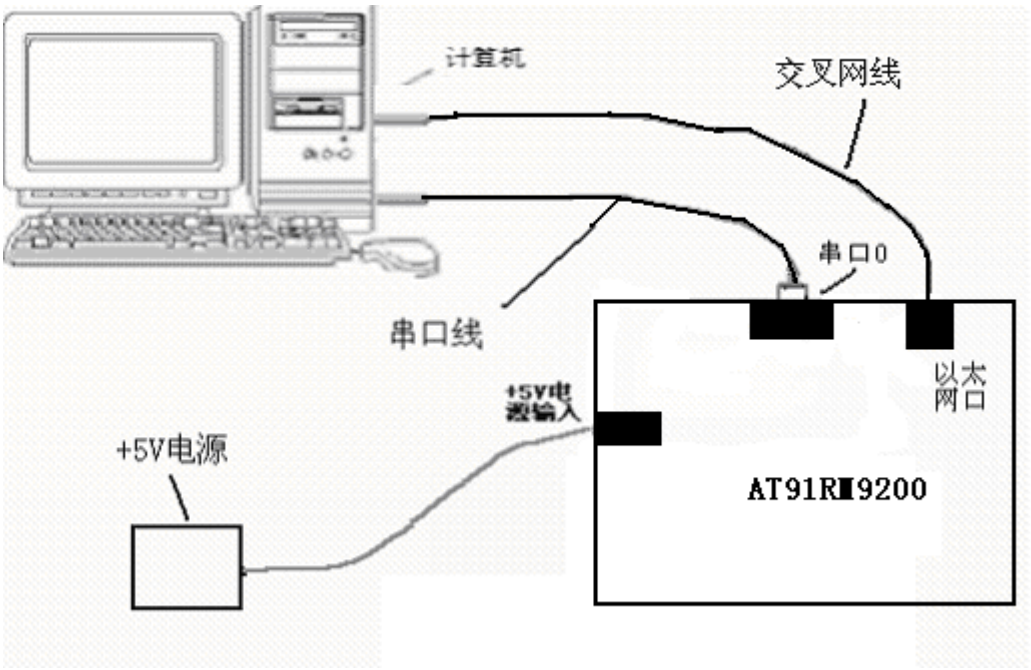


图 1

第二章　运行 Linux 操作系统下的例程

2．1　启动 Uboot 并加载 Linux

首先进行跳线设置，插上 JP1，把 JP2 跳为低电平（low，JP2 在串口附近），则从外部 flash 启动。把它跳到高电平（high），系统会从内部 rom 启动，这时才可以通过 bootrom 把用户程序烧写进 flash，再进行以下操作。

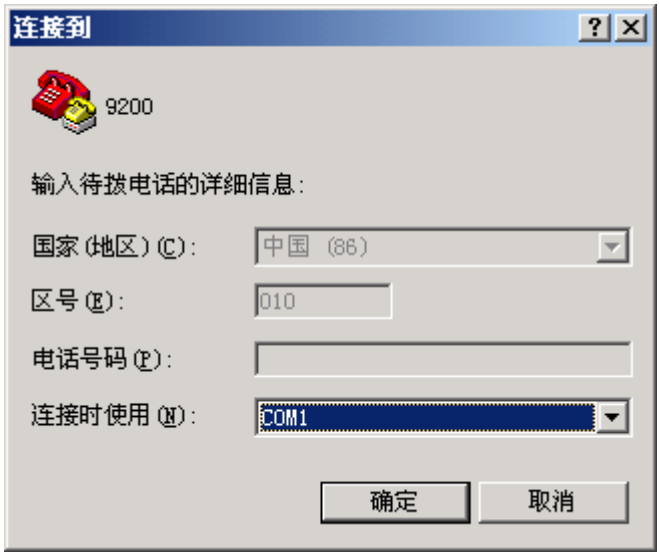
注意：该实验箱在交给用户时，已经由公司测试人员烧写 flash，您只需要将 JP2 的跳线跳为低电平（low），即可从外部 flash 启动。在您初次使用该开发板时，请不要操作步骤 C!

PC 机是通过超级终端（windows）或 minicom（linux）来与实验箱交互通讯的，它为两者间通讯提供操作的界面。下面以通过超级终端与开发板通讯为例说明。

注：超级终端和 minicom 的使用方法见附录。

A、超级终端的建立

按图 1 连接硬件，在 pc 机上打开超级终端，在以下对话框中选择 COM1(如果你使用 COM2 请选择 COM2)，如下图

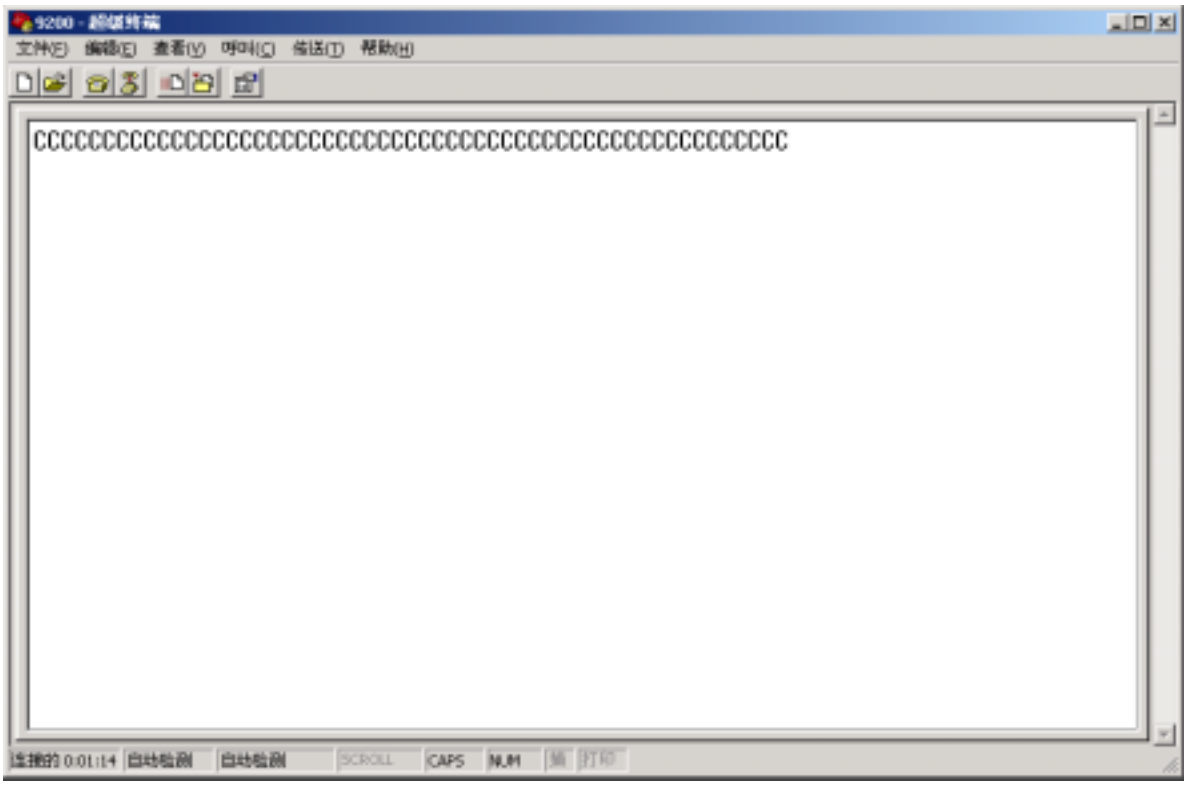


设置串口每秒位数为 115200、数据位为 8 位、奇偶校验为无、停止位为 1，数据流控制为无，如下图：

B、从内部 ROM 启动加载 U-BOOT

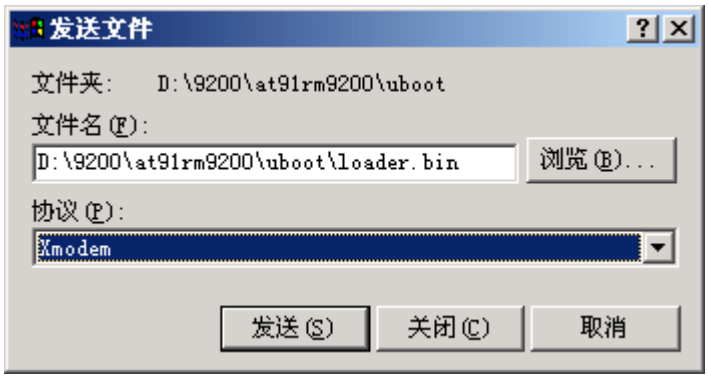
首先，将 JP2 跳为高电平（high）！

给开发板上电并按下复位键让板子复位，然后超级终端中显示 CCCCC，这时，系统是从 AT91RM9200 的内部 ROM 启动：

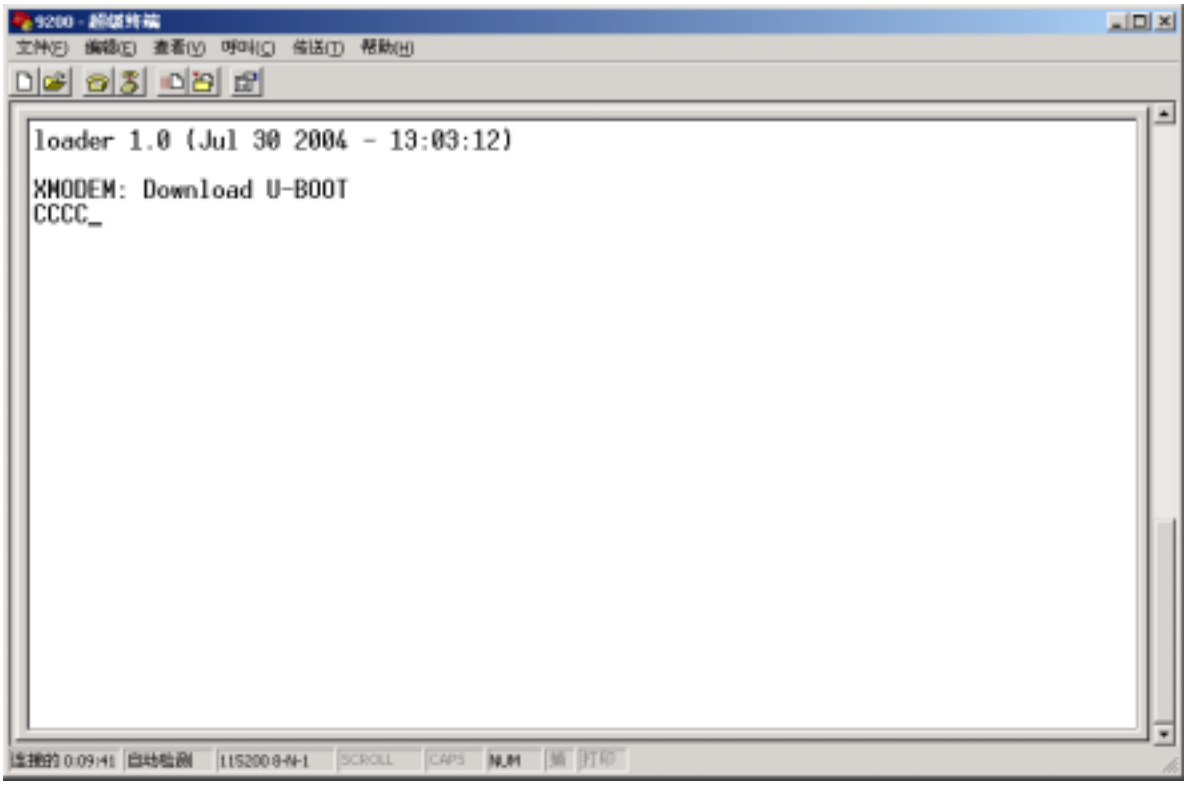


在超级终中选择菜单“ 传送 ”，接着选择菜单“ 发送文件 ”弹出如下对话框，点击“ 浏

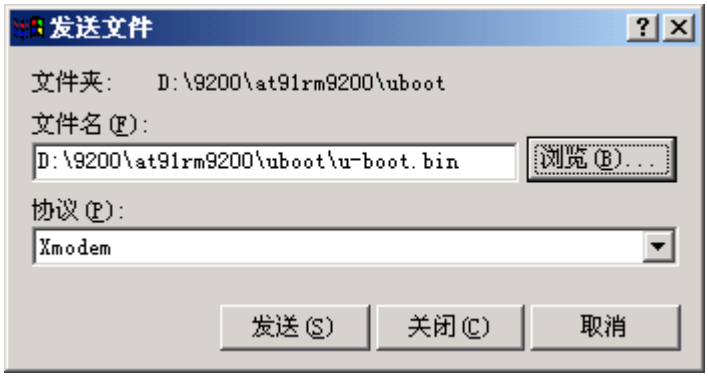
览 ”按钮找到 uboot（您的 flash 芯片容量，对应相应的文件夹）目录下的 loader.bin（您的路径可能和对话框中显示的不一样），协议选择 “ Xmodem ”，如下图，然后点击 “ 发送 ” 按钮发送文件



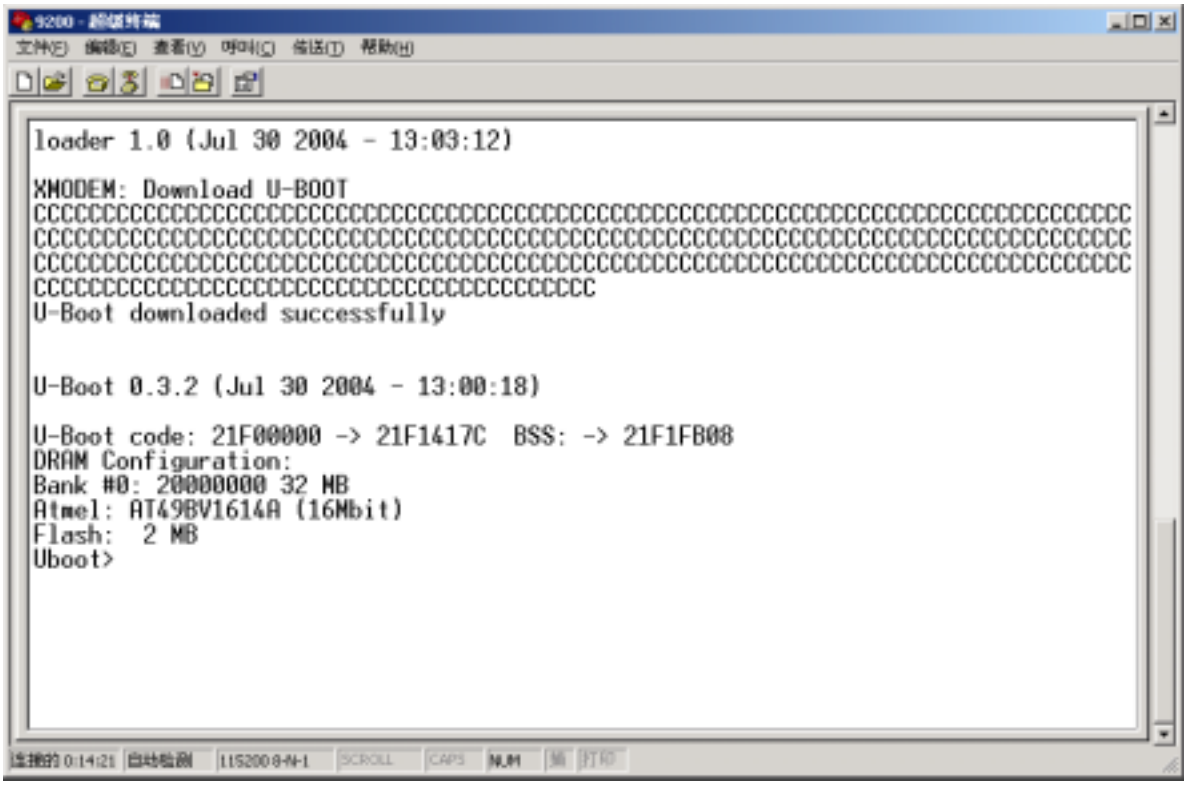
发送完成后会出现如下对话框：



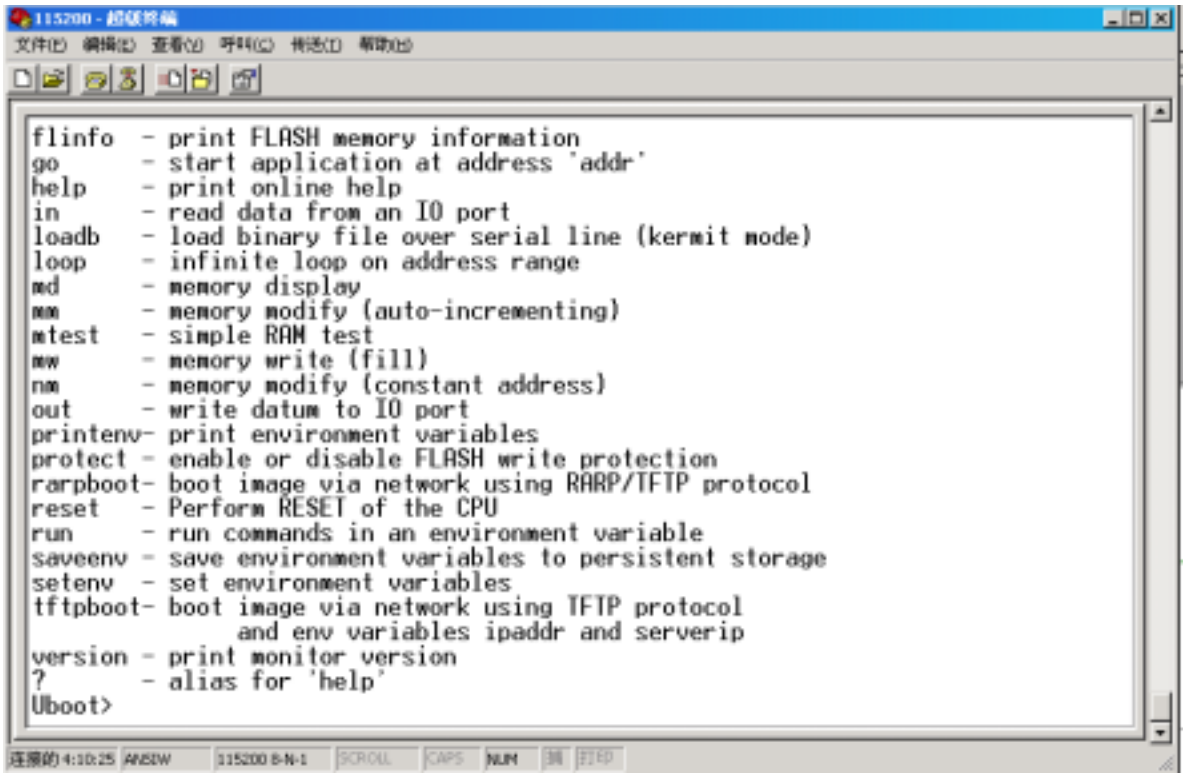
这个时候你继续按上面的提示，在 Xmodem 协议下发送文件 u-boot.bin 文件，如下图



完成后显示提示符 Uboot>，如下对话框所示



这里，我们已经通过 AT91RM9200 的内部 bootrom 将 U-BOOT 下载到 SDRAM 中运行！用户可以使用 help 命令来查看 U-BOOT 下的指令！



C.通过 U-BOOT 烧写用户程序到 flash

下面，我们通过运行刚刚加载的 uboot 程序，把 boot.bin 和 u-boot.gz 文件烧写固

化到 flash 中去。

注意：该步骤在您对此实验板没有完全熟练掌握之前，不要操作，否则后果自负！

首先擦除 FLASH，输入命令

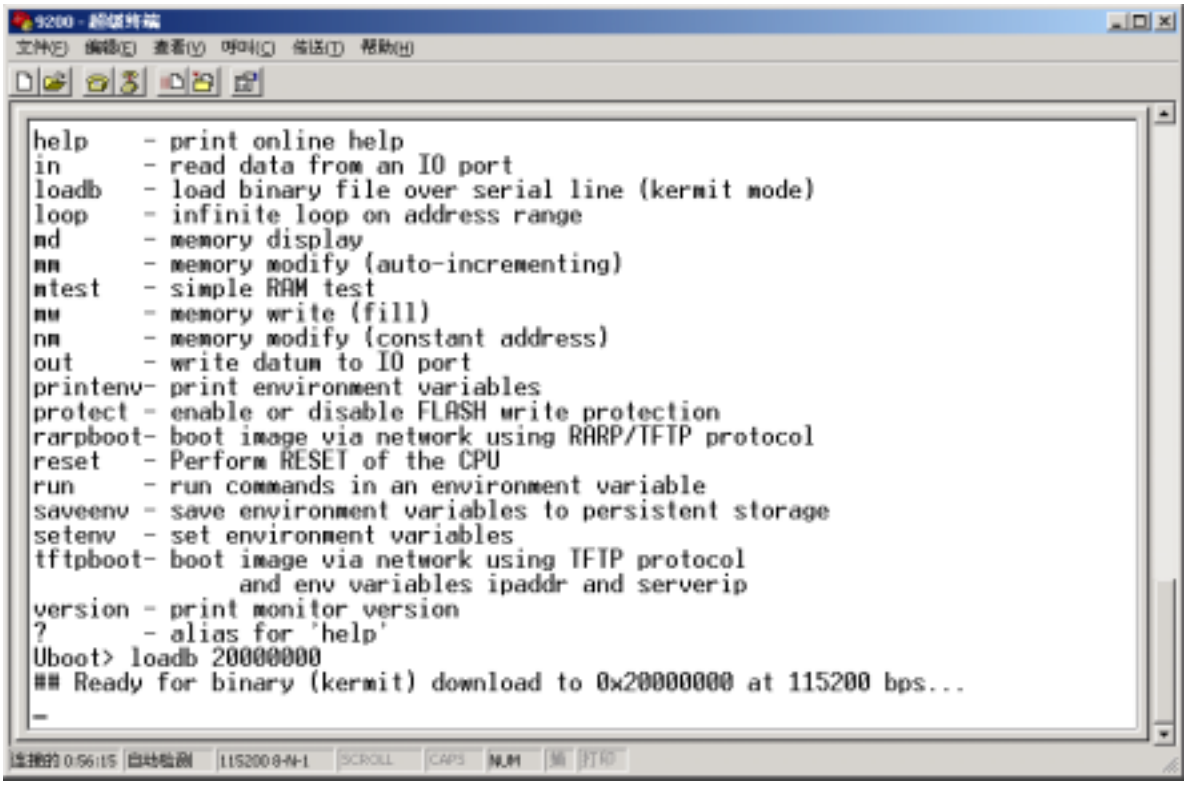
```
Uboot>protect off all
Uboot>erase all
```

等待擦除完毕，出现提示符。

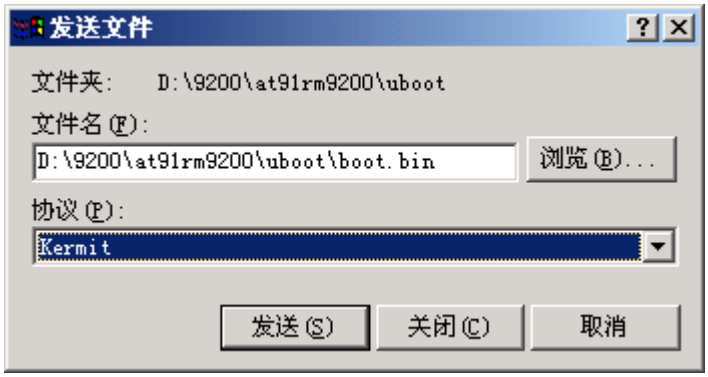
把 boot.bin 和 u-boot.bin.gz 装入 FLASH，在提示符后输入命令

```
Uboot>loadb 20000000
```

出现 Ready for binary (kermit) download to 0x20000000 at 115200 bps...的提示，如下：


help - print online help
in - read data from an IO port
loadb - load binary file over serial line (kermit mode)
loop - infinite loop on address range
md - memory display
mm - memory modify (auto-incrementing)
mtest - simple RAM test
mw - memory write (fill)
nm - memory modify (constant address)
out - write datum to IO port
printenv- print environment variables
protect - enable or disable FLASH write protection
rarpboot- boot image via network using RARP/TFTP protocol
reset - Perform RESET of the CPU
run - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv - set environment variables
tftpboot- boot image via network using TFTP protocol
and env variables ipaddr and serverip
version - print monitor version
? - alias for 'help'
Uboot> loadb 20000000
Ready for binary (kermit) download to 0x20000000 at 115200 bps...

然后使用 Kermit 协议发送文件 boot.bin，如下



发送完 boot.bin 后在提示符下我们输入如下命令

```
Uboot> cp.b 20000000 10000000 ffff
```

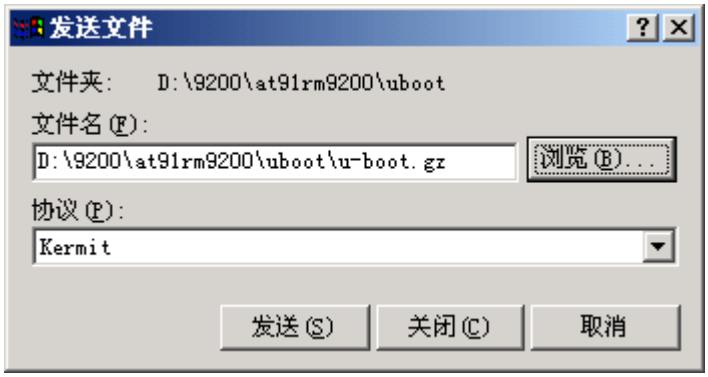
拷贝完以后，我们使用下列命令保护相应的 flash 区域

```
Uboot> protect on all
```

接着我们使用以下命令装入 u-boot.gz ！

```
Uboot>loadb 20000000
```

后出现 Ready for binary (kermit) download to 0x20000000 at 115200 bps...的提示，然后我们然后我们使用协议 Kermit 发送文件 u-boot.bin.gz/u-boot.gz，如下



发送完 u-boot.gz 后在提示符下我们输入如下命令

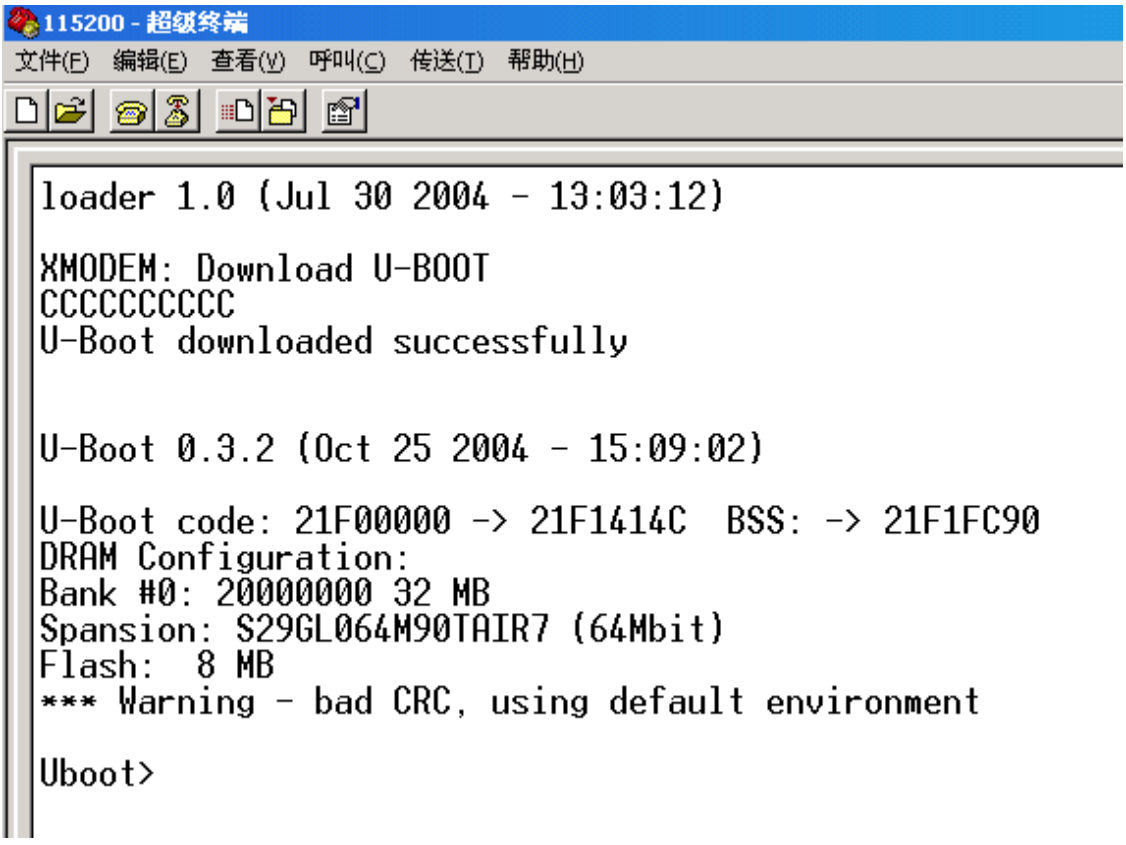
```
Uboot> cp.b 20000000 10020000 ffff
```

拷贝完了以后我们使用下列命令保护相应的 flash 区域

```
Uboot> protect on all
```

执行完以后，会显示保护的扇区数！

到这里，您的 U-BOOT 完全成功烧写到实验箱的 flash 中：



D：linux 操作系统的下载运行

Step1，准备工作：

- 连接网线，采用网线（交叉）直接把实验箱和 PC 机相连，或者采用网线（直

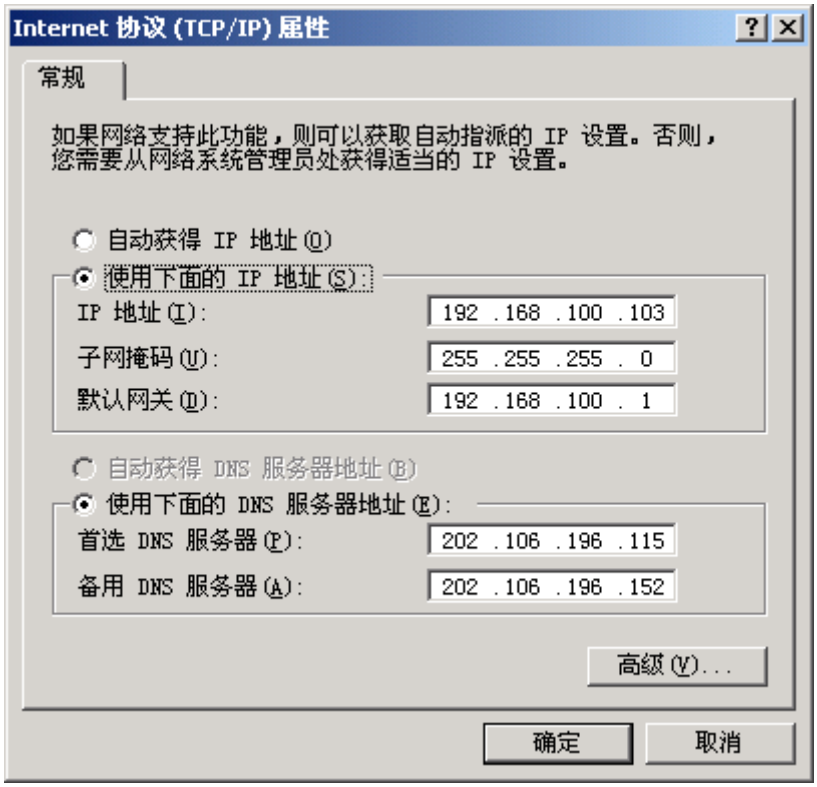
连）接到局域网的 HUB 上

- 把光盘中 tftpserver 文件夹拷贝到根目录下，保证内核影像 uImage 和文件系统影像 tdramdisk 都拷贝在 tftpserver 的目录中。

Step2，Linux 内核下载：

注意：这里涉及到所有的 IP 地址，用户根据自己的网络配置！

- 首先我们把 PC 机的 IP 地址设为 192.168.100.103，子网掩码设为 255.255.255.0，这样才能进行文件传输，如下图



然后双击 tftpserver 目录下的程序 TFTPSRV.EXE 运行它，它是文件下载服务器程序。

- 进行实验箱以太网参数设置

```
Uboot> protect off all                          ；把 flash 写保护去掉
Uboot> setenv ethaddr 12:12:12:12:12:12         ；设置目标板的 mac 地址
Uboot> setenv bootdelay 3                       ；从 uboot 进入 linux 的延时
Uboot> setenv loadaddr 210000000
Uboot> setenv netmask 255.255.255.0         ；设置目标板的子网掩码
Uboot> setenv ipaddr 192.168.0.112              ；设置目标板的 IP 地址
Uboot> setenv serverip 192.168.0.103            ；设置所用 PC 的 IP 地址
Uboot> setenv bootcmd bootm 0x10030000 0x100e0000
Uboot> saveenv                                  ；参数保存，如不保存掉电后须重新设置
Uboot> protect on all                           ；把 flash 写保护
```

- 下载烧写 linux 内核和文件系统

```
Uboot> protect off all
Uboot> tftp 0x20000000 uImage                   ；下载 linux 内核，耗时约 3 分钟
Uboot> cp.b 20000000 10030000 bffff
Uboot> tftp 0x20000000 tdramdisk                ；下载 linux 文件系统,耗时约 15 分钟
```

Uboot> cp.b 20000000 100e0000 5fffff

Uboot> protect on all

注意：这里 uImage 和 tdramdiak 是烧写到 flash 中的文件，而 zImage 和 tdramdisk.gz 是直接加载到 sdram 中运行的代码。用户可以通过以下方式来启动 linux：

Uboot> tftp 0x20000000 zImage

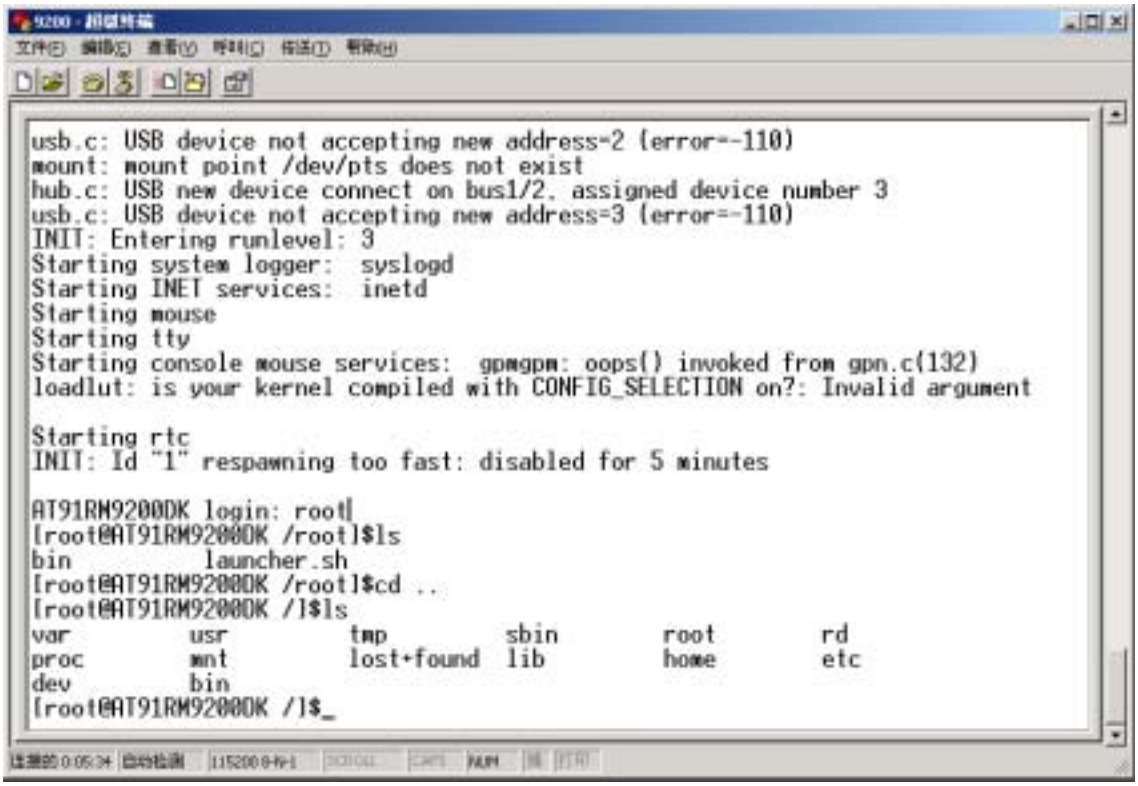
Uboot> tftp 0x20000000 tdramdisk

Uboot> go 0x20008000

将跳线由高电平跳至低电平，成功复位启动运行后，显示提示符 AT91RM9200DK login: 在其后输入用户名 root

AT91RM9200DK login: root

然后按 enter 键后出现[root@AT91RM9200DK /root]$便登陆完成，此后你就可以象在 PC 上使用 linux 一样使用。



2．2 通过 ftp 方式运行 testled 例程

按照上述过程在目标板上启动 linux，并设置好目标板的 ip 地址，然后通过 ftp 把文件下载到目标板上运行！

这里，假设您已经将光盘 linux/demo/testled 目录下的文件拷贝到例程的 linux 服务器中，并且用您的身份登陆能够看见它！其拷贝方法可以参考第四章节内容！

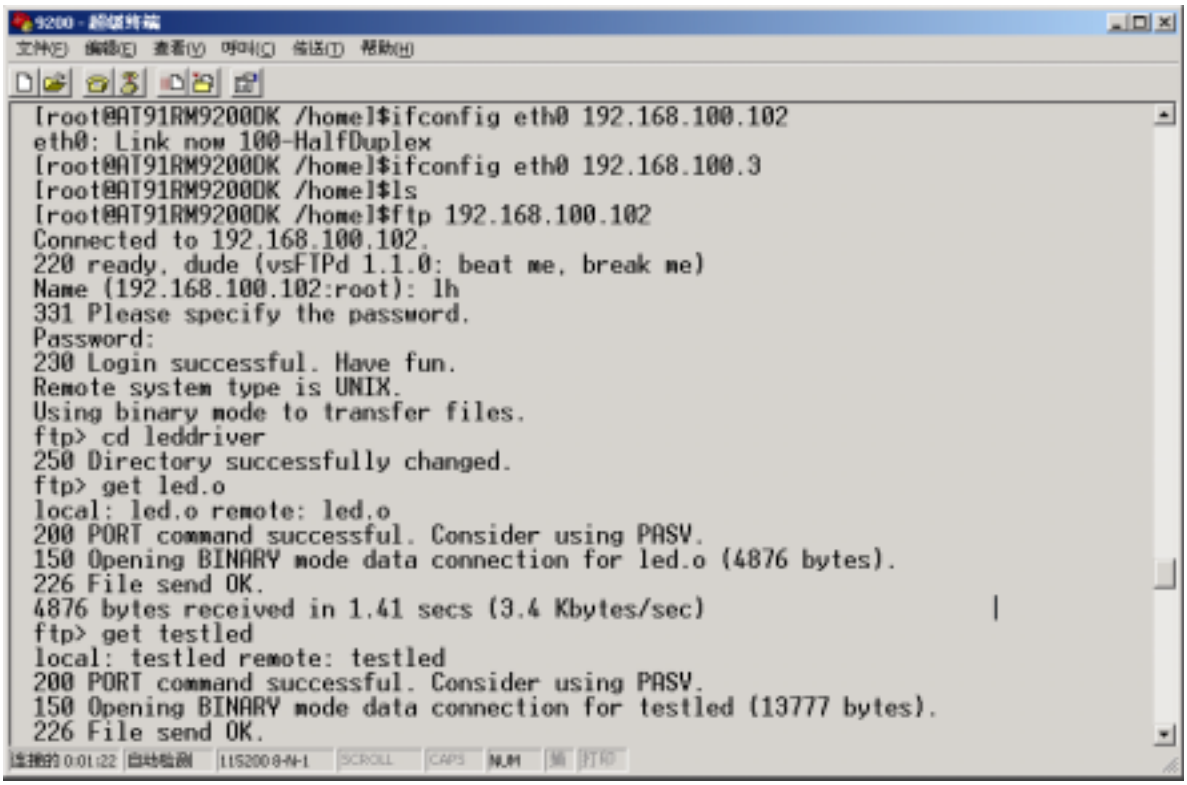
首先进入目标板 linux 下的可读写目录如：

```
# cd /home
```

进入目标板上的/home 目录。

在超级终端中的命令行下敲入如下命令登陆 ftp 服务器（设 ftp 服务器的 ip 地址为 192.168.100.102），显示如下：

```
# ftp 192.168.100.102
```



如上，输入您的 linux 服务器的用户名回车，在 Password 后输入密码后回车，然后可以使用 ls、cd 等命令显示 ftp 服务器上的文件并进入相应的目录。

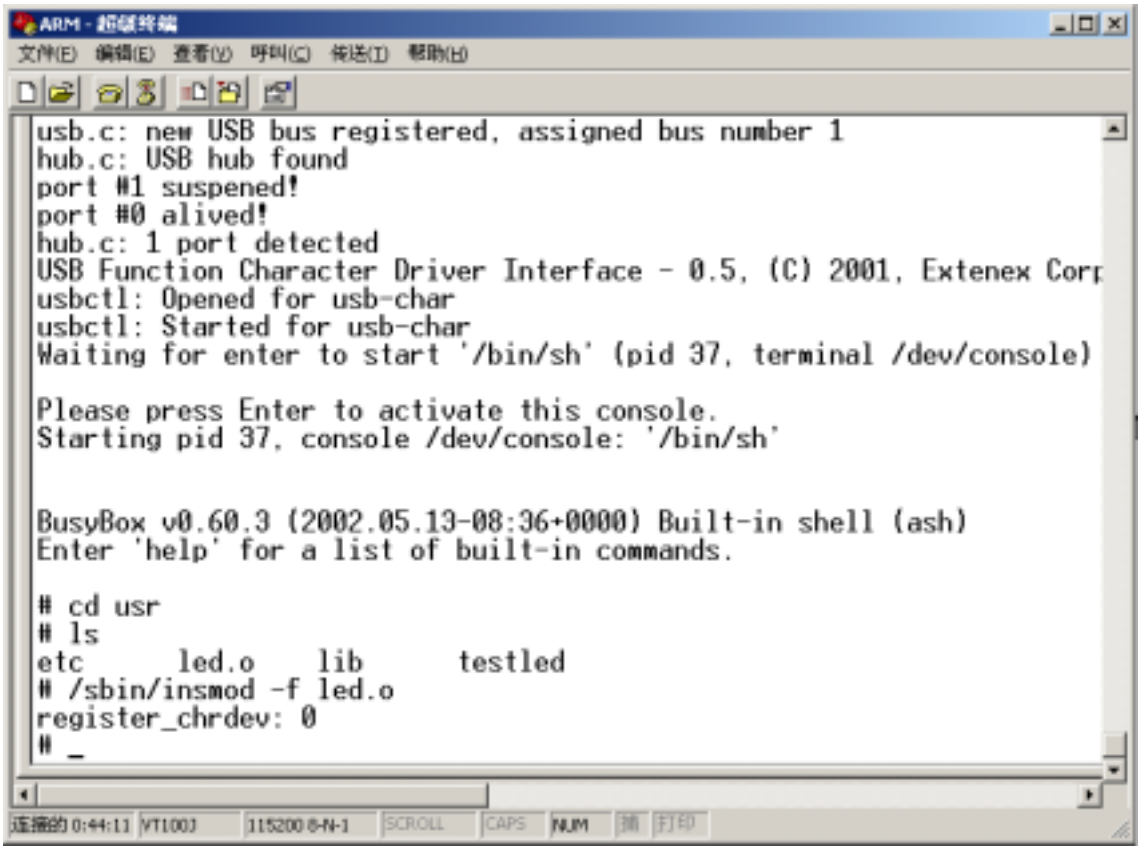
然后使用　get led.o

　　get testled

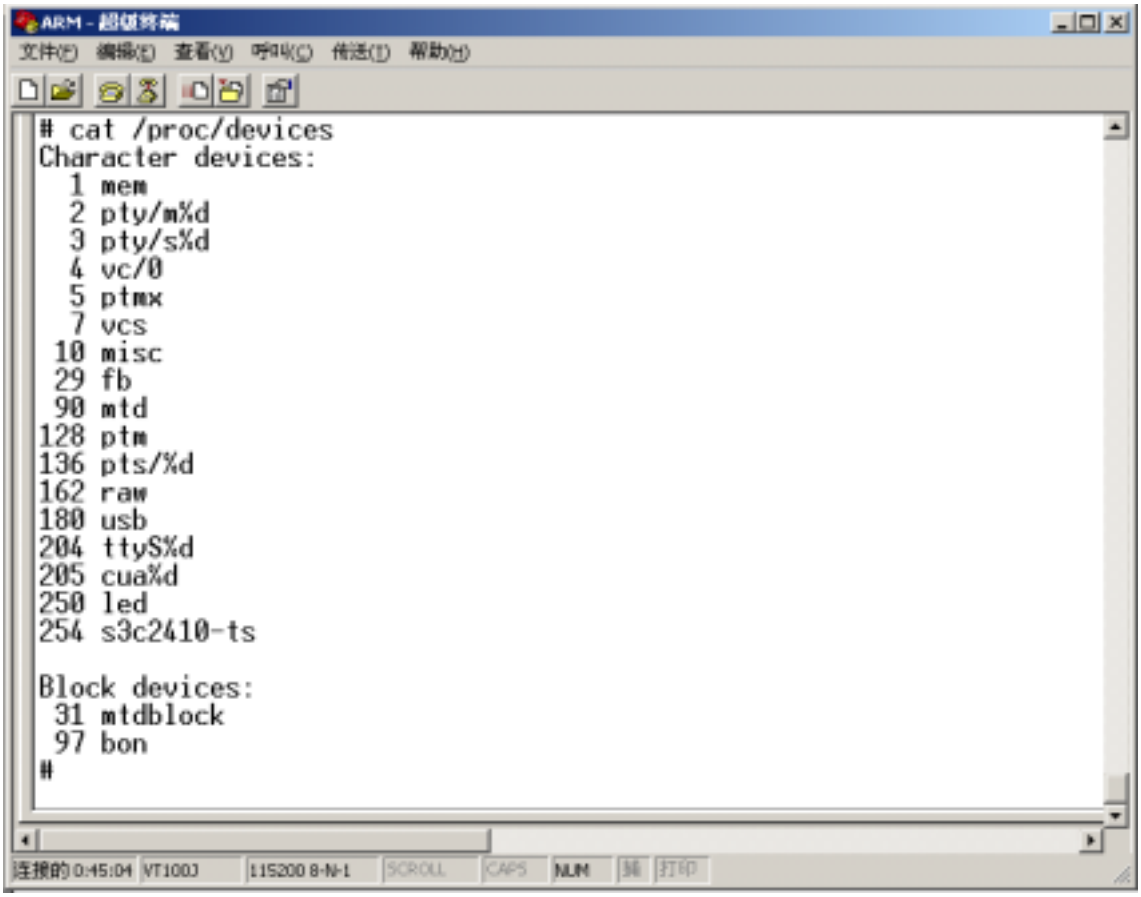
把 testled 文件夹下的文件 led.o 和 testled 下载到目标板上的/home 目录下。

然后执行如下操作：

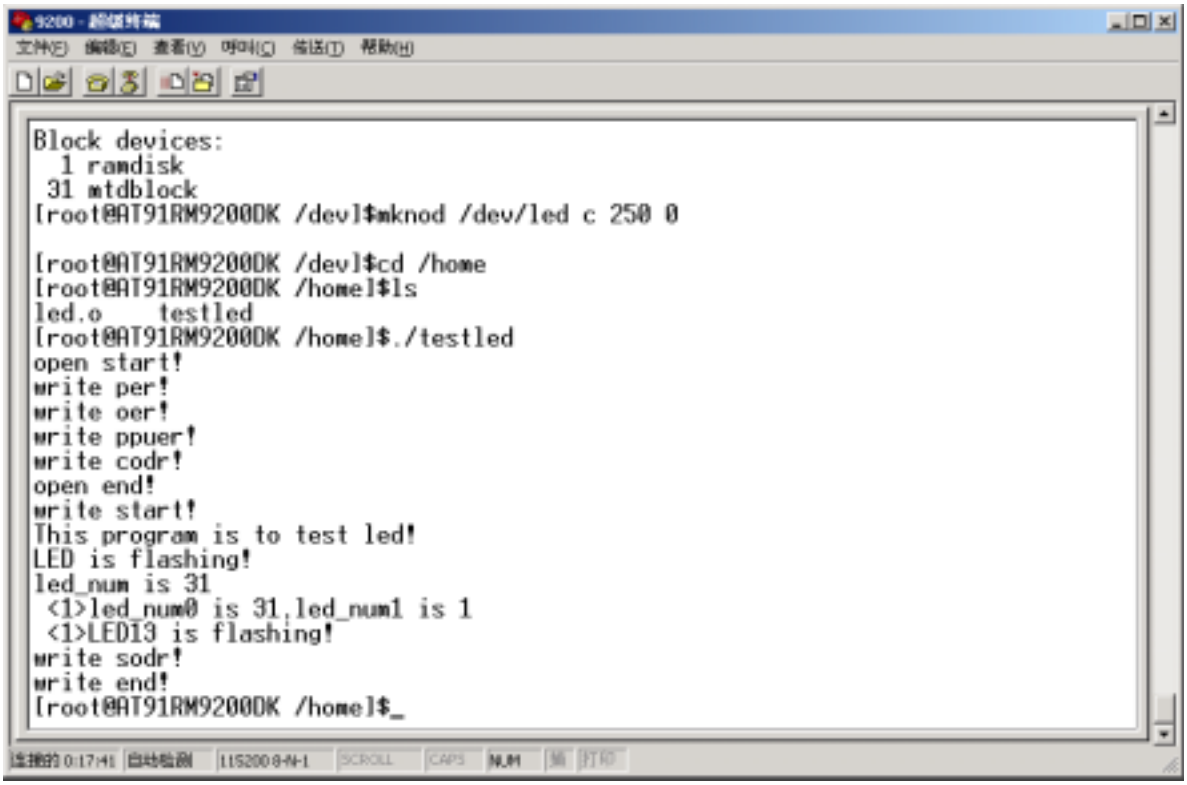
加载模块：键入 `# /sbin/insmod –f led.o`


ARM - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
usb.c: new USB bus registered, assigned bus number 1
hub.c: USB hub found
port #1 suspened!
port #0 alived!
hub.c: 1 port detected
USB Function Character Driver Interface - 0.5, (C) 2001, Extenex Corp
usbctl: Opened for usb-char
usbctl: Started for usb-char
Waiting for enter to start '/bin/sh' (pid 37, terminal /dev/console)
Please press Enter to activate this console.
Starting pid 37, console /dev/console: '/bin/sh'
BusyBox v0.60.3 (2002.05.13-08:36+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
cd usr
ls
etc led.o lib testled
/sbin/insmod -f led.o
register_chrdev: 0
_
连接的 0:44:11 VT100J 115200 8-N-1 SCROLL CAPS NUM 捕 打印

查看 led 的主设备号：键入 `# cat/proc/devices`，如图：

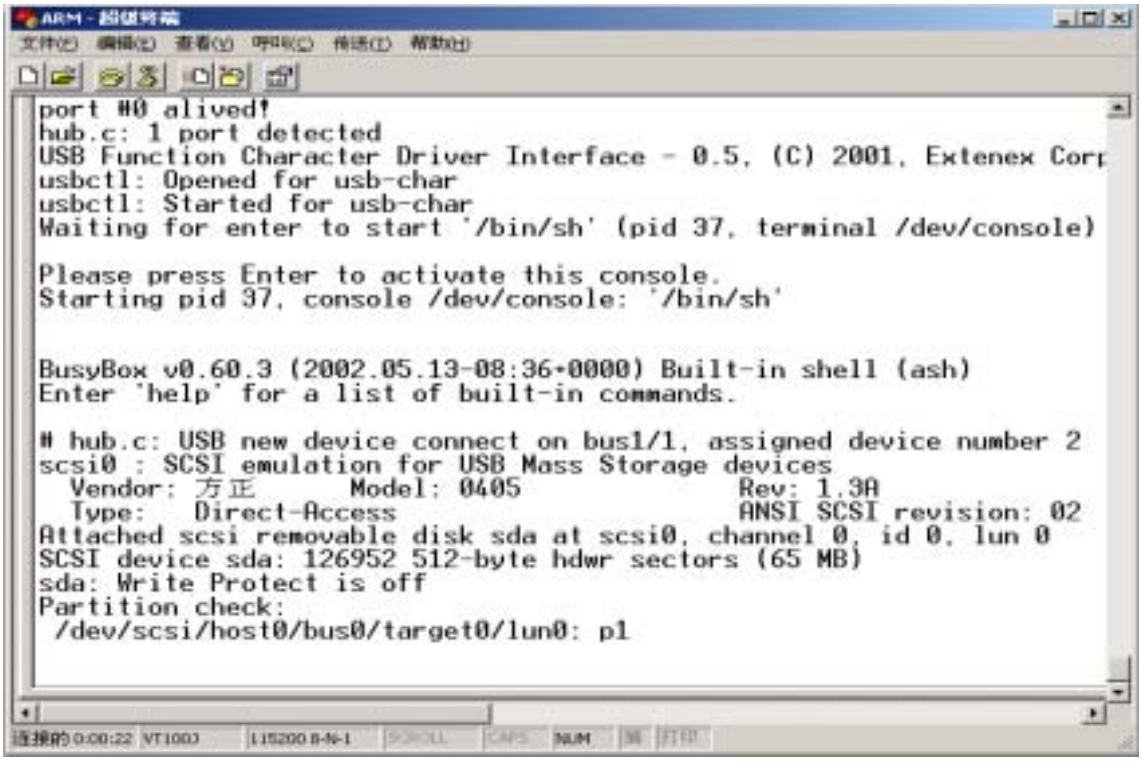


加载模块后，在 Character devices:下比以前多了“ 250 led ” ，也就是加载 LED 模块成功。
建立设备节点：键入 # mknod /dev/led c 250 0（其中的 250 为上一步中查看 led 到的主设备号，应根据实据情况做改动）
运行测试程序：键入 # ./testled
测试时也可不使用示例程序，可采用以下方法：
echo 9 > /dev/led （其中 echo 后可以使用 9、10、12、13、0，0 表示熄灭所有的灯，其余的表示点亮相应的灯）
这时可以看到目标板上 LED13 的灯是亮的，其它的灯都熄灭了，出现如下信息：



2．3 加载卸载优盘实验

启动 linux 后将优盘插在相应的插槽内，出现以下提示信息：

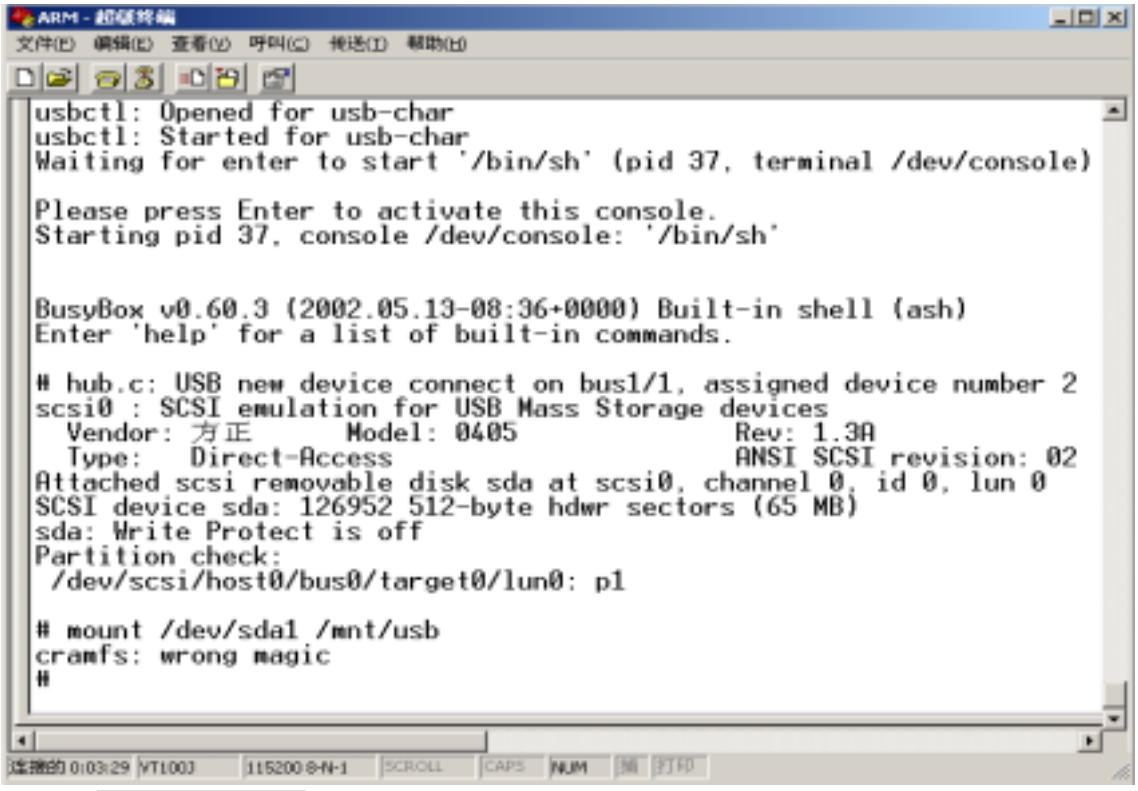

ARM - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
port #0 alived!
hub.c: 1 port detected
USB Function Character Driver Interface - 0.5, (C) 2001, Extenex Corp
usbctl: Opened for usb-char
usbctl: Started for usb-char
Waiting for enter to start '/bin/sh' (pid 37, terminal /dev/console)
Please press Enter to activate this console.
Starting pid 37, console /dev/console: '/bin/sh'
BusyBox v0.60.3 (2002.05.13-08:36+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
hub.c: USB new device connect on bus1/1, assigned device number 2
scsi0 : SCSI emulation for USB Mass Storage devices
Vendor: 方正 Model: 0405 Rev: 1.3A
Type: Direct-Access ANSI SCSI revision: 02
Attached scsi removable disk sda at scsi0, channel 0, id 0, lun 0
SCSI device sda: 126952 512-byte hdwr sectors (65 MB)
sda: Write Protect is off
Partition check:
/dev/scsi/host0/bus0/target0/lun0: p1
连接的 0:00:22 VT100J 115200 8-N-1 SCROLL CAPS NUM 捕 打印

回车后键入：

`# mknod /dev/sda1 b 8 1` （建立设备节点）

`# mount /dev/sda1 /mnt/usb` （加载设备模块）

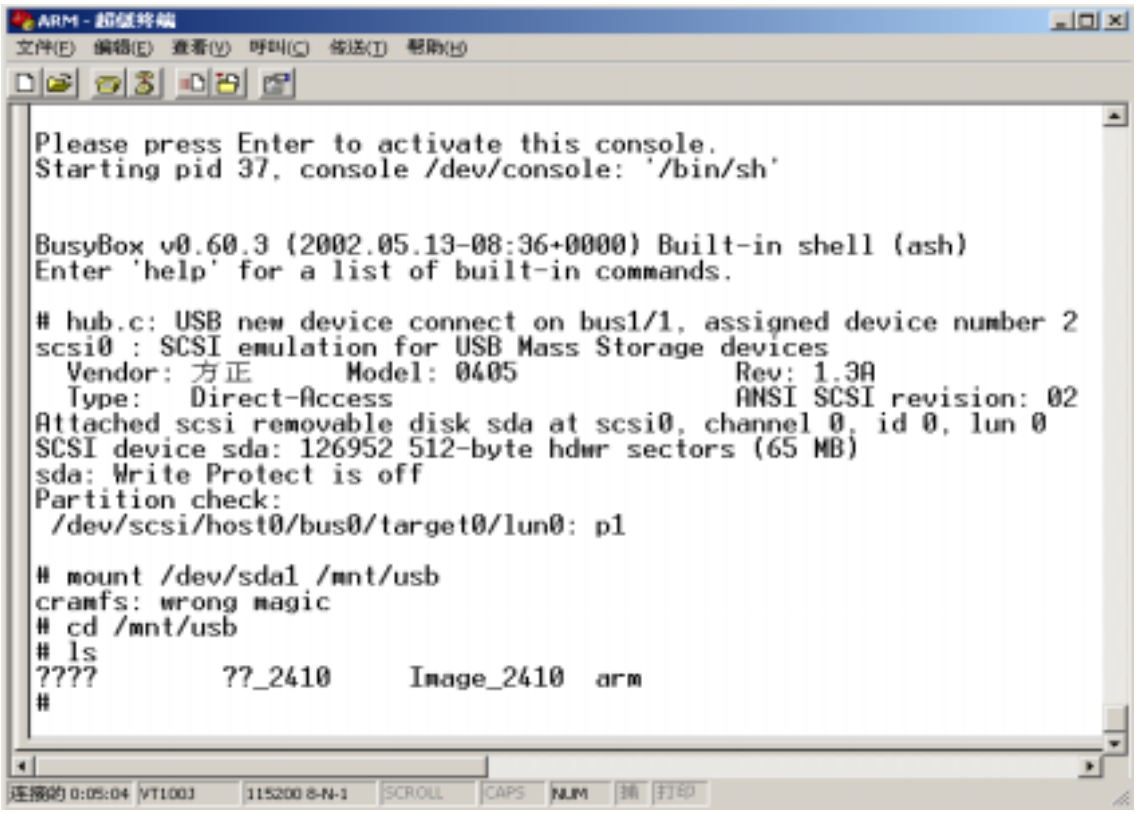
加载优盘到开发板的`/mnt/usb` 目录


ARM - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
usbctl: Opened for usb-char
usbctl: Started for usb-char
Waiting for enter to start '/bin/sh' (pid 37, terminal /dev/console)
Please press Enter to activate this console.
Starting pid 37, console /dev/console: '/bin/sh'
BusyBox v0.60.3 (2002.05.13-08:36+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
hub.c: USB new device connect on bus1/1, assigned device number 2
scsi0 : SCSI emulation for USB Mass Storage devices
Vendor: 方正 Model: 0405 Rev: 1.3A
Type: Direct-Access ANSI SCSI revision: 02
Attached scsi removable disk sda at scsi0, channel 0, id 0, lun 0
SCSI device sda: 126952 512-byte hdwr sectors (65 MB)
sda: Write Protect is off
Partition check:
/dev/scsi/host0/bus0/target0/lun0: p1
mount /dev/sda1 /mnt/usb
cramfs: wrong magic
#
连接的 0:03:29 VT100J 115200 8-N-1 SCROLL CAPS NUM 捕 打印

键入 `# cd /mnt/usb`

`# ls`

查看优盘中的内容，显示如下：



上图显示的就是优盘的内容。

键入 `# cd /`

`# umount /mnt/usb`

就可以卸载优盘。如图：

2．4　优盘读写测试实验

启动 linux 后将优盘插在相应的插槽内，出现提示信息后：
回车键入：

```
# mknod /dev/sda1 b 8 1 （建立设备节点）
# mount /dev/sda1 /mnt/usb （加载设备模块）
```

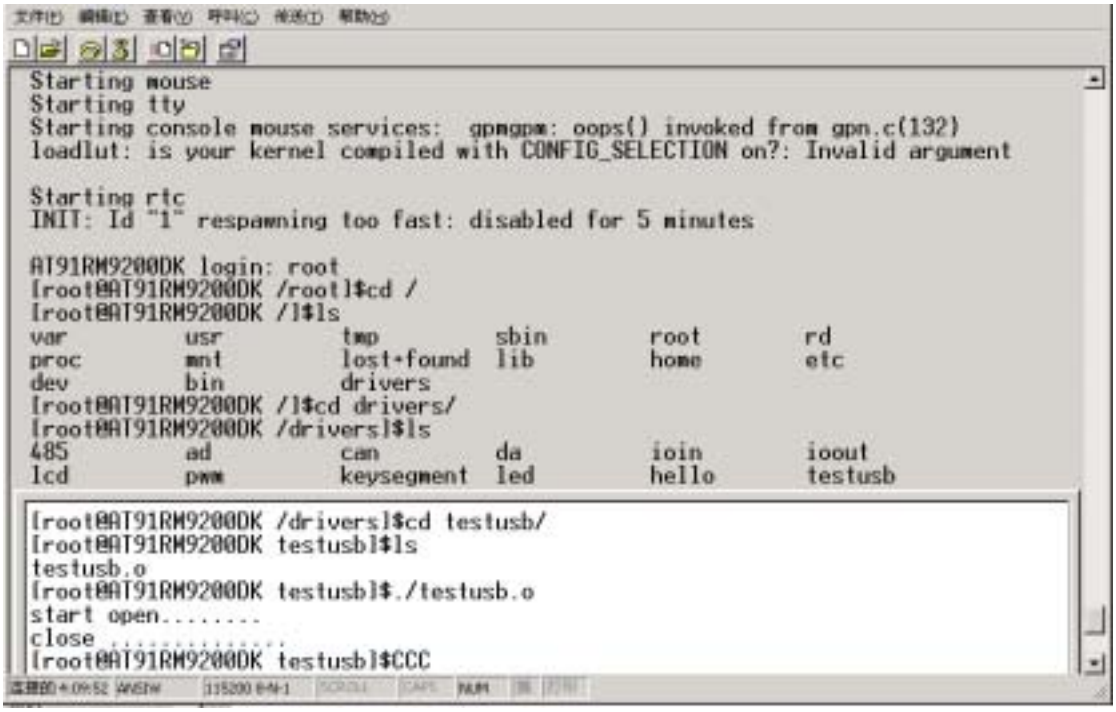
加载优盘到开发板的/mnt/usb 目录。
键入：

```
# cd /mnt/usb
# ls
```

可以查看优盘中的文件。
然后进入实验系统 linux 根目录下的 drivers/testusb 目录，运行该文件夹下的可执行文件。

```
# cd /drivers/testusb
# ls
# ./testusb.o
```

该 usb 测试程序代码所实现的功能是：在用户优盘的根目录下新建一个 winsilicon.txt 的文本文件，打开该文件后写进数据“ 1234567890 ”，然后保存关闭。如下图所示：



用户可以到优盘的根目录下查看 winsilicon.txt 文件是否存在，并查看它的内容！
此外，在光盘 linux\demo\testusb 目录下，存在有 testusb 文件的源代码。用户可以编辑该源代码，利用 makefile 文件在 linux 下重新编译生成目标代码，加载到文件系统中。

第三章　交叉编译环境和程序调试环境的建立

3．1　建立交叉编译环境

宿主机建议安装 Red Hat Linux 8.0 以上版本

提示：当图片信息中的路径与文字路径不一致时，以文字路径为准！

以下以在 SHELL 模式下安装编译器和源代码为例来介绍安装过程：
首先以 root 身份进入系统。
把安装光盘放入光驱中。执行以下命令：

```
mount /dev/cdrom /mnt/cdrom
```

这是装载cdrom 。如果您的CDROM 已经在安装RedHat 的时候已经默认安装，以上命令请不要执行，请直接进入CDROM 所在目录。

```
mkdir /usr/local/arm
mkdir /usr/ws-430
cd /
cp /mnt/cdrom/linux/tools/cross-2.95.3.tar.bz2  /usr/local/arm/
cp /mnt/cdrom/linux/tools/linux-2.4.19-ws.tar.gz  /usr/ws-430/
cd /usr/ws-430
tar zxvf linux-2.4.19-ws.tar.gz
cd /usr/local/arm/
tar -jxvf cross-2.95.3.tar.bz2
```

程序执行完后，Linux 源代码和编译环境都已经建立起来了，Linux 源码在/usr/ws-430/linux-2.4.19-rmk7/文件夹下。当你需要使用的时候应该指出编译器的路径，如果你不想每次都指明，你也可以通过使用下面设置环境变量命令 PATH＝/usr/local/arm/2.95.3/bin:$PATH 来把开发工具加到你的路径下。

3．2　Linux 内核编译

首先以 root 身份进入系统,进入/usr/ws-430/linux-2.4.19-rmk7，然后完成下列步骤，将会在/usr/ws-430/linux-2.4.19-rmk7/arch/arm/boot/下得到内核的映像文件 zImage。

用户可以做修改，然后重新编译新内核，然后下载到目标板上去。这时如果你在图形界面下，你可以按下 Ctrl＋Alt＋F1 进入字符模式，然后敲入以下的命令。

在 shell 下键入如下命令：

```
make at91rm9200dk_config
make oldconfig
make menuconfig
```

内核配置选项：该命令执行完毕后生成文件.config，它保存这个配置信息。下一次再做 make menuconfig 的时候将产生新的.config 文件，在 general 选项里的倒数第二个选项中应该把默认的内核命令字符串设为：console=ttys0,115200

initrd=0x21000000（注：这个指明了文件系统的起始位置，内核影像中的程序执行完了以后会跳动这个位置运行）,5145728(注：这个值指定文件系统的大小，如果过小会出现 ran out of 的错误) root=/dev/ram rw。

键入命令：make dep

键入命令：make zImage

通过各个目录的 Makefile 文件进行，会在各目录下生成一大堆目标文件。

上述步骤完成后，就完成了对 linux 源码的编译工作。整个编译过程视计算机运行速度而定，大约需要几分钟到十几分钟左右。

下部 Linux 程序开发提高篇

第四章 Linux 驱动、应用程序的编写

4．1 编写 Linux 应用程序

以编写 Hello World 为例介绍编写应用程序的过程。

<1> 在宿主机 red hat Linux 环境下用 vi 或其它文本编辑器编辑一个 test.c 文件。

其内容如下：

```
#include<stdio.h>
int main(void)
{
  printf(“ hello world!\n ”);
  return 0;

}
```

存盘、退出编辑器。

在/usr/ws-430 目录下建立一个 char 目录 ： mkdir /usr/ws-430/char

将 test.c 拷贝到 /usr/ws-430/char 目录下

执行以下命令：

```
/usr/local/arm/2.95.3/bin/arm-linux-gcc —o test.o test.c
```

键入 chmod 755 test.o

给 test.o 文件加上可执行属性。

此时，在目标板上可以执行的文件 test.o 已经编译完成。

<2> 在开发板上执行 test.o 文件

启动开发板的 Linux 系统，完成开发板和 linux 服务器的 ftp 连接。
首先进入目标板 linux 下的可读写目录如：

```
# cd /home
```

进入目标板上的/home 目录。
在超级终端中的命令行下敲入如下命令登陆 ftp 服务器（设 ftp 服务器的 ip 地址为 192.168.100.102），显示如下：

```
# ftp 192.168.100.102
```

输入您的 linux 服务器的用户名回车，在 Password 后输入密码后回车，然后可以使用 ls、cd 等命令显示 ftp 服务器上的文件并进入相应的目录。
键入
get test.o
把文件 test.o 下载到目标板上的/home 目录下。
执行
quit
退出 ftp 服务！
cd /home
进入目标板的 home 目录。
使用 ls 等查看/home 中的内容，发现 test.o 已经下传到目标板！

然后执行如下操作：# ./test.o

在屏幕上出现 test.c 程序要输出的“ hello world! ”，说明程序已经正确执行。显示如图：

```
#
#
#
#
#
#
#
#
#
#
# cd tmp
# ls
char                                root_china
date                                root_china.tar.bz2
image_new                           root_english.tar.bz2
linux-2.4.18-rmk7-pxa1-mz4.tar.gz   run.sh
mkcramfs                            usr
root                                vivi
root.tar.bz2                        vivi-20030629.tar.bz2
# cd char
# ls
int_timer0  led         test.c      test.o
# ./test.o
hello world!
# _
```

4．2　调试 Linux 驱动程序

以调试已经编好的 LED 闪烁的驱动程序为例，说明调试过程。Led 的驱动程序和应用程序放在计算机的/usr/ws-430/char/led 目录下。

注：该驱动程序和应用程序的源码可以在附带光盘的例程中找到。

启动主机的 Linux 系统

键入 `# /usr/ws-430/char/led`

`# ls`

可以看到如下图所示的 led 目录下的内容：

```
BusyBox v0.60.3 (2002.05.13-08:36+0000) Built-in shell (as
Enter 'help' for a list of built-in commands.

# ifconfig eth0 192.168.100.112
# mount 192.168.100.102:/usr/ws-350 /tmp
# cd tmp
# ls
char                                root_china
date                                root_china.tar.bz2
image_new                           root_english.tar.bz2
linux-2.4.18-rmk7-pxa1-mz4.tar.gz   run.sh
mkcramfs                            usr
root                                vivi
root.tar.bz2                        vivi-20030629.tar.bz2
# cd char
# ls
int_timer0  led         test.c      test.o
# cd led
# ls
Makefile   led.c      led.o      testled    testled.c
#
```

编译写好的驱动程序和应用程序：
键入 `# make`
执行 Makefile 文件，即：执行对驱动程序和应用程序的编译。

加载模块：键入 `# /sbin/insmod –f led.o`

```
BusyBox v0.60.3 (2002.05.13-08:36+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# ifconfig eth0 192.168.100.112
# mount 192.168.100.102:/usr/ws-350 /tmp
# cd tmp
# ls
char                                root_china
date                                root_china.tar.bz2
image_new                           root_english.tar.bz2
linux-2.4.18-rmk7-pxa1-mz4.tar.gz   run.sh
mkcramfs                            usr
root                                vivi
root.tar.bz2                        vivi-20030629.tar.bz2
# cd char
# ls
int_timer0  led         test.c      test.o
# cd led
# ls
Makefile   led.c      led.o      testled    testled.c
# /sbin/insmod -f led.o
register_chrdev: 0
# _
```

建立设备节点：键入 `# mknod /dev/led c 250 0`

```
BusyBox v0.60.3 (2002.05.13-08:36+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# ifconfig eth0 192.168.100.112
# mount 192.168.100.102:/usr/ws-350 /tmp
# cd tmp
# ls
char                                root_china
date                                root_china.tar.bz2
image_new                           root_english.tar.bz2
linux-2.4.18-rmk7-pxa1-mz4.tar.gz   run.sh
mkcramfs                            usr
root                                vivi
root.tar.bz2                        vivi-20030629.tar.bz2
# cd char
# ls
int_timer0  led         test.c      test.o
# cd led
# ls
Makefile   led.c      led.o      testled    testled.c
# /sbin/insmod -f led.o
register_chrdev: 0
# mknod /dev/led c 250 0
# _
```

运行测试程序：键入 `# ./testled`

```
char                                    root_china
date                                    root_china.tar.bz2
image_new                               root_english.tar.bz2
linux-2.4.18-rmk7-pxa1-mz4.tar.gz       run.sh
mkcramfs                                usr
root                                    vivi
root.tar.bz2                            vivi-20030629.tar.bz2
# cd char
# ls
int_timer0  led         test.c      test.o
# cd led
# ls
Makefile   led.c      led.o      testled    testled.c
# /sbin/insmod -f led.o
register_chrdev: 0
# mknod /dev/led c 250 0
# ./testled

This program is to test led!
LED is flashing!
```

第五章　Linux 下设备驱动程序的开发

5．1 基于设备接口的 Linux 驱动程序

LINUX 的驱动程序是基于标准的设备接口，即 LINUX 系统对设备的操作都会转换成对抽象的标准文件操作。本文主要讲解基于 LINUX-2.4.18 的字符设备的驱动程序开发流程。

1　基于 LINUX-2.4.18 的文件操作结构定义。

file_operations 定义于<linux/fs.h>文件中,随着内核的不断升级，file_operations 结构也越来越大，不同版本的内核会稍有不同。

```
/*
 * NOTE:
 * read, write, poll, fsync, readv, writev can be called
 *   without the big kernel lock held in all filesystems.
 */
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
loff_t *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t
*, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                unsigned long, unsigned long, unsigned long);
     };
```

file_operations 结构中的成员几乎全部是函数指针，所以实质上就是函数跳转表，每个进程对设备的操作，都会根据设备的 major,minor 设备号，转换成对

file_operations 结构的访问。

常用的操作包含以下几种：

struct module *owner;

结构指针 owner 指向拥有该结构的模块，内核使用该指针维护模块使用计数。

loff_t (*llseek) (struct file *, loff_t, int);

方法 llseek 用来修改文件当前位置，并将新位置作为结果返回。loff_t 是在 LINUX 中定义的长偏移量。出错时返回负值。

size_t (*read) (struct file *, char *, size_t, loff_t *);

方法 read 从设备中读取数据，函数返回非负值表示成功读取的字节数。

size_t (*write) (struct file *, const char *, size_t, loff_t *);

方法 write 向设备中写入数据，函数返回非负值表示成功写入的字节数。

unsigned int (*poll) (struct file *, struct poll_table_struct *);

方法 poll 是系统调用 select 和 poll 的后端实现，用这两个系统调用来查询设备是否可以读写，或是否处于某种状态。如果 poll 为空，则驱动设备会被认为即可读又可写，返回值是一个状态掩码。

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

方法 ioctl 为系统调用 ioctl 提供了一种执行设备特定命令的方法。

int (*mmap) (struct file *, struct vm_area_struct *);

方法 mmap 将设备内存映射到进程地址空间。

int (*open) (struct inode *, struct file *);

方法 open 用于打开设备，open 操作是对设备的第一个操作，如果 open 方法为空，则设备始终打开成功。

int (*fasync) (int, struct file *, int);

方法 fasync 仅用于网路文件系统的代码中。

int (*release) (struct inode *, struct file *);

方法 release 在关闭文件时调用

在自己的驱动程序中，首先要根据驱动程序的功能，完成 file_operations 结构中函数实现，当不需要的函数接口，如 fsync，fasync，lock 等，可以直接在 file_operations 结构中初始化为 NULL。file_operations 变量会在驱动程序初始化时，注册到系统内部，当操作系统对设备操作时会调用驱动程序注册的 file_operations 结构中的函数指针。

2 设备驱动的初始化函数

设备驱动的初始化函数主要完成的功能是：

a 对驱动程序管理的硬件进行必要的初始化。

b 初始化设备驱动相关的参数。

c 在内核中注册设备。

Linux 内核通过设备的主设备号和从设备号来访问设备驱动，每个驱动程序都有唯一的主设备号。驱动程序注册设备时可以由系统分配一个唯一的设备号，但每次分配主设备号都有可能不同；也可以在驱动程序中指定好主设备号。当 Linux 内核支持 proc 文件系统时，可以在系统启动后，在/proc/devices 确认各个驱动程序的设备号。

d 如果设备需要 IRQ 支持，则要注册中断。

```
int request_irq(unsigned int irq, void (*handler)(int, void *, struct
pt_regs *),
                unsigned long irq_flags, const char * devname, void
*dev_id)；
```

函数注册中断服务函数,其中：

irq ：中断号

handler ：中断响应处理函数

irq_flags ：中断管理选项掩码

devname ：设备名称

dev_id ：中断共享时使用

e 其他初始化工作。

3 初始化程序如何声明。

当驱动程序是内核的一部分，则要按如下方式声明，注意不能缺少__init

```
int __init chr_driver_init(void)；
```

则在系统启动的时候会由内核调用 chr_driver_init，完成的驱动程序的初始化。

当驱动程序是以模块的形式编写分，则要按如下方式声明，当运行 insmod 命令插入模块时会调用 init_module 函数完成初始化工作。

```
int init_module (void)
```

5．2 基于模块化的字符设备驱动框架

1 模块化驱动程序框架 ：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/slab.h>
#include <linux/proc_fs.h>
#include <linux/config.h>
#include <linux/version.h>
#include <linux/init.h>
#include <asm/uaccess.h>
```

```
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,3,1)
#define init_MUTEX(s) do { *(s) = MUTEX; } while(0)
#endif

int init_module (void)
{
    return 1;
}
void cleanup_module(void)
{
    return;
}
```

init_module()在运行 insmod 命令后由系统调用，完成驱动模块的初始化工作。 Cleanup_module()在运行 rmmod 命令后由系统调用，完成驱动模块的卸载时的清除工作。

2 一个 LED 字符设备驱动程序的分析

```
#define  __KERNEL__
#define    MODULE
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <asm/uaccess.h>
#include <linux/string.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/interrupt.h>
#include <linux/ptrace.h>
#include <linux/ioport.h>
#include <linux/in.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include <linux/string.h>
#include <asm/bitops.h>
#include <asm/io.h>
#include <linux/errno.h>
#include <linux/tqueue.h>
#include <linux/wait.h>
#include <asm/irq.h>
#include <asm/arch/hardware.h>
```

```
#include <asm/arch/irqs.h>
#include <linux/devfs_fs_kernel.h>

#define QUERY_FREQ (HZ/50)
//下面定义寄存器的物理地址，有些没用到，可以去掉，但是为了完整，我们都定义了
// ========== Register definition for PIOC peripheral ==========
#define AT91C_PIOC_IFDR ((AT91_REG *)     0xFFFFF824)
// (PIOC) Input Filter Disable Register
#define AT91C_PIOC_ODR  ((AT91_REG *)     0xFFFFF814)
// (PIOC) Output Disable Registerr
#define AT91C_PIOC_ABSR ((AT91_REG *)     0xFFFFF878)
// (PIOC) AB Select Status Register
#define AT91C_PIOC_SODR ((AT91_REG *)     0xFFFFF830)
// (PIOC) Set Output Data Register
#define AT91C_PIOC_IFSR ((AT91_REG *)     0xFFFFF828)
// (PIOC) Input Filter Status Register
#define AT91C_PIOC_CODR ((AT91_REG *)     0xFFFFF834)
// (PIOC) Clear Output Data Register
#define AT91C_PIOC_ODSR ((AT91_REG *)     0xFFFFF838)
// (PIOC) Output Data Status Register
#define AT91C_PIOC_IER  ((AT91_REG *)     0xFFFFF840)
// (PIOC) Interrupt Enable Register
#define AT91C_PIOC_IMR  ((AT91_REG *)     0xFFFFF848)
// (PIOC) Interrupt Mask Register
#define AT91C_PIOC_OWDR ((AT91_REG *)     0xFFFFF8A4)
// (PIOC) Output Write Disable Register
#define AT91C_PIOC_MDDR ((AT91_REG *)     0xFFFFF854)
// (PIOC) Multi-driver Disable Register
#define AT91C_PIOC_PDSR ((AT91_REG *)     0xFFFFF83C)
// (PIOC) Pin Data Status Register
#define AT91C_PIOC_IDR  ((AT91_REG *)     0xFFFFF844)
// (PIOC) Interrupt Disable Register
#define AT91C_PIOC_ISR  ((AT91_REG *)     0xFFFFF84C)
// (PIOC) Interrupt Status Register
#define AT91C_PIOC_PDR  ((AT91_REG *)     0xFFFFF804)
// (PIOC) PIO Disable Register
#define AT91C_PIOC_OWSR ((AT91_REG *)     0xFFFFF8A8)
// (PIOC) Output Write Status Register
#define AT91C_PIOC_OWER ((AT91_REG *)     0xFFFFF8A0)
// (PIOC) Output Write Enable Register
#define AT91C_PIOC_ASR  ((AT91_REG *)     0xFFFFF870)
// (PIOC) Select A Register
#define AT91C_PIOC_PPUSR ((AT91_REG *)    0xFFFFF868)
// (PIOC) Pad Pull-up Status Register
#define AT91C_PIOC_PPUDR ((AT91_REG *)    0xFFFFF860)
```

```
// (PIOC) Pull-up Disable Register
#define AT91C_PIOC_MDSR ((AT91_REG *)     0xFFFFF858)
// (PIOC) Multi-driver Status Register
#define AT91C_PIOC_MDER ((AT91_REG *)     0xFFFFF850)
// (PIOC) Multi-driver Enable Register
#define AT91C_PIOC_IFER ((AT91_REG *)     0xFFFFF820)
// (PIOC) Input Filter Enable Register
#define AT91C_PIOC_OSR  ((AT91_REG *)     0xFFFFF818)
// (PIOC) Output Status Register
#define AT91C_PIOC_OER  ((AT91_REG *)     0xFFFFF810)
// (PIOC) Output Enable Register
#define AT91C_PIOC_PSR  ((AT91_REG *)     0xFFFFF808)
// (PIOC) PIO Status Register
#define AT91C_PIOC_PER  ((AT91_REG *)     0xFFFFF800)
// (PIOC) PIO Enable Register
#define AT91C_PIOC_BSR  ((AT91_REG *)     0xFFFFF874)
// (PIOC) Select B Register
#define AT91C_PIOC_PPUER ((AT91_REG *)    0xFFFFF864)
// (PIOC) Pull-up Enable Register

//typedef volatile unsigned int AT91_REG;
// Hardware register definition
#define GPIOLEN    0X20
#define GPIOC 0xFFFFF800
//GPIOC REGISTER BASE ADDRESS
#define PER_OFFSET 0X00
#define OER_OFFSET 0X10
#define SODR_OFFSET         0X30
#define CODR_OFFSET         0X34
#define PPUER_OFFSET    0X64

int key_open(struct inode * inode, struct file * filp);
int key_release(struct inode * inode, struct file * filp);
ssize_t key_read(struct file * filp, char * buf, size_t count, loff_t * l);
ssize_t key_write(struct file * filp,const char * buf,size_t count,loff_t * l);
unsigned char  led_val=0x0,led_num[2]={'0','0'};
unsigned int leddata=0;
static void * gpioc_base;

//这个驱动实现了 open 、release 和 write 功能，以下结构定义了实现以上功能的函数
struct file_operations key_fops = {
    open:           key_open,
    release:        key_release,
    read:           key_read,
    write:          key_write,
```

```
};

//下面的代码一行写不完时放到了下一行，没有用连接符连接，请注意一下
int key_open(struct inode * inode, struct file * filp)
{
     printk("<1>open start!\n");
     MOD_INC_USE_COUNT;
     gpioc_base=ioremap_nocache(GPIOC,GPIOLEN);
    //得到寄存器的虚拟地址
    //Step1 Enable pio,把与 led 灯相连的管脚设为 gpio。
    writel((unsigned      int)(1<<15)|(unsigned      int)(1<<14)|(unsigned      int)(1)|(unsigned
    int)(1<<1),gpioc_base+PER_OFFSET);
     printk("<1>write per!\n");
     //Step2 Enable Output，使相应的管脚输出使能
     writel((unsigned      int)(1<<15)|(unsigned      int)(1<<14)|(unsigned      int)(1)|(unsigned
    int)(1<<1),gpioc_base+OER_OFFSET);
     printk("<1>write oer!\n");
     //Step3 Enable Pull-up ，把相应的管脚设为上拉
     writel((unsigned      int)(1<<15)|(unsigned      int)(1<<14)|(unsigned      int)(1)|(unsigned
    int)(1<<1),gpioc_base+PPUER_OFFSET);
     printk("<1>write ppuer!\n");
     //Step4 Exchange light PIO14 and PIO15 For 10 Times，把四个灯都熄灭
     writel((unsigned      int)(1<<15)|(unsigned      int)(1<<14)|(unsigned      int)(1)|(unsigned
    int)(1<<1),gpioc_base+CODR_OFFSET);
     printk("<1>write codr!\n");
     printk("<1>open end!\n");
     return 0;
}

int key_release(struct inode * inode,struct file * filp)
{
     MOD_DEC_USE_COUNT;
     iounmap(gpioc_base);
     return 0;
}

//用 led_val 的四位分别记录相应的 led 灯的状态，读取相应的位即可知道那个灯是亮的
ssize_t key_read(struct file * filp, char * buf, size_t count, loff_t * l)
{
       if (led_val & 0x1)
         led_num[0]='9';
       else
         if (led_val & 0x2)
              {led_num[0]='0';led_num[1]='1';}
         else
```

```
            if (led_val & 0x4)
                {led_num[0]='2';led_num[1]='1';}
            else
                if (led_val & 0x8)
                    {led_num[0]='3';led_num[1]='1';}
                else
                    {led_num[0]='0';led_num[1]='0';}
    copy_to_user(buf, &led_num, 1);
    return 0;
}
```

//读取用户态下的应用程序传来的 buf 的值，通过判断相应的字符组织输出数据，然后往相应的输出数据寄存器中写数据点亮相应的管脚

```
ssize_t key_write(struct file * filp, const char * buf, size_t count, loff_t * l)
{
     printk("<1>write start!\n");
      leddata=0;
     copy_from_user(&led_num,buf,2);
     //printk("<1>!\n");

      printk("<1>This program is to test led!\n");
      printk("<1>LED is flashing!\n");
      printk("<1>led_num is %s\n ",led_num);
      printk("<1>led_num0 is %s,led_num1 is %s\n ",&led_num[0],&led_num[1]);
      if(led_num[0]=='9')
      {

        printk("<1>LED9 is flashing!\n");
        led_val=led_val | 0x1;
        leddata=leddata | (unsigned int)(1<<14);
      }
      if(led_num[0]=='0' && led_num[1]=='1')
      {
        printk("<1>LED10 is flashing!\n");
        led_val=led_val | 0x2;
        leddata=leddata | (unsigned int)(1<<15);
      }
      if(led_num[0]=='2' && led_num[1]=='1')
      {
        printk("<1>LED12 is flashing!\n");
        led_val=led_val | 0x4;
        leddata=leddata | (unsigned int)(1<<1);
      }
      if(led_num[0]=='3' && led_num[1]=='1')
      {
```

```
          printk("<1>LED13 is flashing!\n");
          led_val=led_val | 0x8;
          leddata=leddata | (unsigned int)(1);
        }
     writel(leddata,gpioc_base+SODR_OFFSET);
     printk("<1>write sodr!\n");
     if(led_num[0]=='0' && led_num[1]=='0')
     writel((unsigned      int)(1<<15)|(unsigned      int)(1<<14)|(unsigned      int)(1)|(unsigned
int)(1<<1),gpioc_base+CODR_OFFSET);
     printk("<1>write end!\n");
     return 0;
}

int init_module(void)
{
     unsigned int result = 0;
     result = register_chrdev(250, "led", &key_fops);

     if (result < 0) {
          printk("<1>int_timer0_dev: can't get major number 250\n");
          unregister_chrdev(250,"led");
          return result;
     }
     printk("<1>register_chrdev: %d\n", result);
     return 0;

}

void cleanup_module(void)
{
     unregister_chrdev(250,"led");
}
```

3 如何安装驱动模块

```
# insmod  mod.o        ;安装模块
# rmmod   mod          ;卸载模块
# lsmod                ;浏览系统已经安装的模块
```

第六章　如何创建文件系统

下面我们介绍在一个建好的文件系统的基础上创建自己的文件系统
解开压缩的文件系统
gunzip ramdisk.gz
影像文件挂装
mount –o loop ramdisk /mnt/newramdisk　(注：newranmdisk 是新建的目录)
进入 newramdisk 目录进行操作，随意的增减文件
cd /mnt/newramdisk (注：进入目录后敲入命令随意的增减文件)
退出 newramdisk 目录后卸装文件系统
umount /mnt/newramdisk
压缩文件系统，生成新的文件系统影像
gzip –C –v9 ramdisk > ./newramdisk

附　录

附录一 Linux 常用命令

（基于 REDHAT LINUX8.0）
基本命令：
ls：显示当前目录下的所有文件和目录。
ls –a ：可以看到隐藏的文件，如以.开头的文件。
pwd：显示当前目录路径。
ps：列举当前TTY 下所有进程
ps –A：列举所有进程
cd 目录名：进入目录
mkdir 目录名：创建目录
rmdir 目录名：删除空目录
rm –rf 目录名：强行删除整个目录内容（无法恢复），其中f 表示强制不进行提示，r 表示目录递归。
LINUX 下的文件和目录是区分大小写的。

TAB键：用于文件目录匹配搜索
例如用户要进入目录/usr，只需敲入：cd /u 然后按下TAB 键，则SHELL 会自动匹配找到usr 目录，这样就不必完全键入剩余的usr 字符，这个功能在访问名字很长的文件和目录时非常有效，可以大大提供键盘输入的速度，极为方便。

Man：在线获取帮助信息（类似于VC中的MSDN）
man，即manunal，是UNIX 系统手册的电子版本。根据习惯，UNIX 系统手册通常分为不同的部分（或小节，即section），每个小节阐述不同的系统内容。目前的小节划分如下：
命令：普通用户命令
系统调用：内核接口
函数库调用：普通函数库中的函数
特殊文件：/dev 目录中的特殊文件
文件格式和约定：/etc/passwd 等文件的格式
游戏；
杂项和约定：标准文件系统布局、手册页结构等杂项内容
系统管理命令；
内核例程：非标准的手册小节。
手册页一般保存在/usr/man 目录下，其中每个子目录（如man1,man2, ..., manl, mann）包含不同的手册小节。使用man 命令查看手册页：

Ipconfig：用于配置主机网络信息
例如修改主机IP地址为192.168.2.32：
ifconfig eth0 192.168.2.32
ifconfig –a 显示当前网络配置信息

压缩/解压缩：
LINUX 的软件一般是以.gz 或.tar 或者.tar.gz 结尾的。前者是由gzip压缩的，后者是先用tar 归档，在用gzip 压缩而成的。
以.gz 结尾的为压缩文件，用命令：gzip -d filename 来解压，得到的文件在当前目录中，但已没有了.gz。
以.tar 结尾的为归档文件，用命令：tar -xvf filename 来展开，生成的文件与源文件在同一目录中，只是少了.tar。
以.tar.gz 结尾的文件最常见，可直接用命令：gzip -cd filename| tar xfv 来解开

vi(m)用法
vi 是Linux/Unix 世界里极为普遍的全屏幕文本编辑器，几乎可以说任何一台Linux/Unix 机器都会提供这个软件。
vi 有三种状态，即编辑方式、插入方式和命令方式。
在命令方式下，所有命令都要以:开始，所键入的字符系统均作命令来处理，如:q代表退出，:w 表示存盘。当你进入vi 时，会首先进入命令方式（同时也是编辑方式）。按下I就进入插入方式，用户输入的可视字符都添加到文件中，显示在屏幕上。按下ESC 就可以回到命令状态（同时也是编辑方式）。
编辑方式和命令方式类似，都是要输入命令，但它的命令不要以:开始，它直接接受键盘输入的单字符或组合字符命令，例如直接按下u 就表示取消上一次对文件的修改，相当于WINDOWS 下的Undo 操作。
编译方式下有一些命令是要以/开始的，例如查找字符串就是：/string 则在文件中匹配查找string 字符串。在编辑模式下按下:就进入命令方式。
基本命令解释：

◆ 光标命令
k、j、h、l——上、下、左、右光标移动命令。虽然您可以在Linux中使用键盘右边的4 个光标键，但是记住这4 个命令还是非常有用的。这4 个键正是右手在键盘上放置的基本位置。
nG——跳转命令。n 为行数，该命令立即使光标跳到指定行。
Ctrl+G——光标所在位置的行数和列数报告。
w、b——使光标向前或向后跳过一个单词。

◆ 编辑命令
i、a、r——在光标的前、后以及所在处插入字符命令(i=insert、a=append、r=replace)。
cw、dw——改变(置换)/删除光标所在处的单词的命令(c=change、d=delete)。
x、d$、dd——删除一个字符、删除光标所在处到行尾的所有字符以及删除整行的命令。

◆ 查找命令
---- /string、?string——从光标所在处向后或向前查找相应的字符串的命令。

◆ 拷贝复制命令

---- yy、p——拷贝一行到剪贴板或取出剪贴板中内容的命令。

常用操作：

无论是开启新档或修改旧文件，都可以使用vi，所需指令为：$ vi filemane 如果文件是新的，就会在荧幕底部看到一个信息，告诉用户正在创建新文件。如果文件早已存在，vi 则会显示文件的首廿四行， 用户可再用光标（cursor）上下移动。

—指令i 在光标处插入正文
—指令I 在一行开始处插入正文
—指令a 在光标後追加正文
—指令A 在行尾追加正文
—指令o 在光标下面新开一行
—指令O 在光标上面新开一行

在插入方式下，不能打入指令，必需先按〈Esc〉键，返回命令方式。假若户不知身处何态，也可以按〈Esc〉键，不管处於何态，都会返回命令方式。

在修改文件时，如何存档及退出指定文件都非常重要。在vi内，行使存档或退出的指令时，要先按冒号（:），改变为命令方式，用户就可以看见在荧幕左下方，出现冒号（:），显示vi 已经改为指令态，可以进行存档或退出等工作。

:q!放弃任何改动而退出vi，也就是强行退出
:w 存档
:w! 对于只读文件强行存档
:wq 存档并退出vi
:x 与wq 的工作一样
:zz 与wq 的工作一样删除正文

删除或修改正文都是利用编辑方式，故此，下面所提及的指令只需在编辑方式下，直接键入指令即行。

—x 删除光标处字符（Character）
—nx 删除光标处後n 个字符
—nX 删除光标处前n 个字符
—ndw 删除光标处下n 个单词（word）
—dd 删除整行
—d$或D 删除由光标至该行最末
—u 恢复前一次所做的删除

当使用vi 修改正文，加减字符时，就会采用另一组在编辑方式下操作的指令。

— r char 由char 代替光标处的字符
—Rtext〈Esc〉由text 代替光标处的字符
—cwtext〈Esc〉由text 取代光标处的单词
—Ctext〈Esc〉由text 取代光标处至该行结尾处
—cc 使整行空白，但保留光标位置，让你开始打入
—如删除指令一样，在指令前打入的数，表示执行该指令多少次。要检索文件，必需在编辑方式下进行。
—／str〈Return〉向前搜寻str 直至文件结尾处
—?str〈Return〉往後搜寻str 直至文件开首处
—n 同一方向上重复检索

—N 相反方向上重复检索
—vi 缠绕整个文件，不断检索，直至找到与模式相匹配的下一个出现。

全程替换命令：
:%s/string1/string2/g 在整个文件中替换“ string1 ”成“ string2 ”。
如果要替换文件中的路径：使用命令“ :%s#/usr/bin#/bin#g ”可以把文件中所有路径/usr/bin 换成/bin。也可以使用命令“ :%s/\/usr\/bin/\/bin/g ”实现，其中“ \ ”是转义字符，表明其后的“ / ”字符是具有实际意义的字符，不是分隔符。同时编辑2 个文件，拷贝一个文件中的文本并粘贴到另一个文件中：
命令如下：
---- vi file1 file2
---- yy 在文件1 的光标处拷贝所在行
---- :n 切换到文件2 (n=next) 或者按ctrl+ww，就在两个文件间切换。
---- p 在文件2 的光标所在处粘贴所拷贝的行
---- :n 切换回文件1
将文件中的某一部分修改保存到临时文件，例如仅仅把第20～59 行之间的内容存盘成文件/tmp/1，我们可以键入如下命令：
---- vi file
---- :20,59w /tmp/1
如果要在 vi 执行期间，转到shell 执行，使用惊叹号（!）执行系统指令，例如在vi 期间，列出当前目录内容，可以键入：
:!ls
另一方面，用户可以在主目录中创建.exrc 环境文件，用set打入选项，每次调用vi 时，就会读入.exrc 中的指令与设置。下面是.exrc 环境文件的实例：
set wrapmarging＝8
set showmode
set autoindent
软、硬盘及光驱的使用
在Linux 中对其他硬盘逻辑分区、软盘，光盘的使用与我们通常在DOS与Windows 中的使用方法是不一样的，不能直接访问，因为在Linux中它们都被视为文件，因此在访问使用前必须使用装载命令mount 将它们装载到系统的/mnt 目录中来，使用结束，必须进行卸载。命令格式如下：
mount －t 文件系统类型设备名装载目录
文件类型常用的有：
msdos dos 分区文件
ext2 Linux 的文件系统
swap Linux swap 分区或swap 文件
iso9660 安装CD-ROM 的文件系统
vfat 支持长文件名的dos 分区
hpfs OS/2 分区文件系统
设备名是指要装载的设备的名称，如软盘、硬盘、光盘等，软盘一般为/dev/fd0 fd1，硬盘一般为/dev/hda hdb ，硬盘逻辑分区一般为期hda1 hda2…等等，光盘一般为/dev/hdc。在装载前一般要在/dev/mnt目录下建立一个空的目录，如软盘为floppy，硬盘分区为其盘符如c、d等等，光盘为cd-rom，使用命令：

```
mount  –t msdos /dev/fd0 /mnt/floppy   装载一个mddos 格式的软盘
mount  –t ext2 /dev/fd0 /mnt/floppy    装载一个Linux 格式的软盘
mount  –t vfat /dev/hda1 /mnt/c 装载Windows98 格式的硬盘分区
mount  –t iso9660 /dev/hdc /mnt/cd-rom 装载一个光盘
```

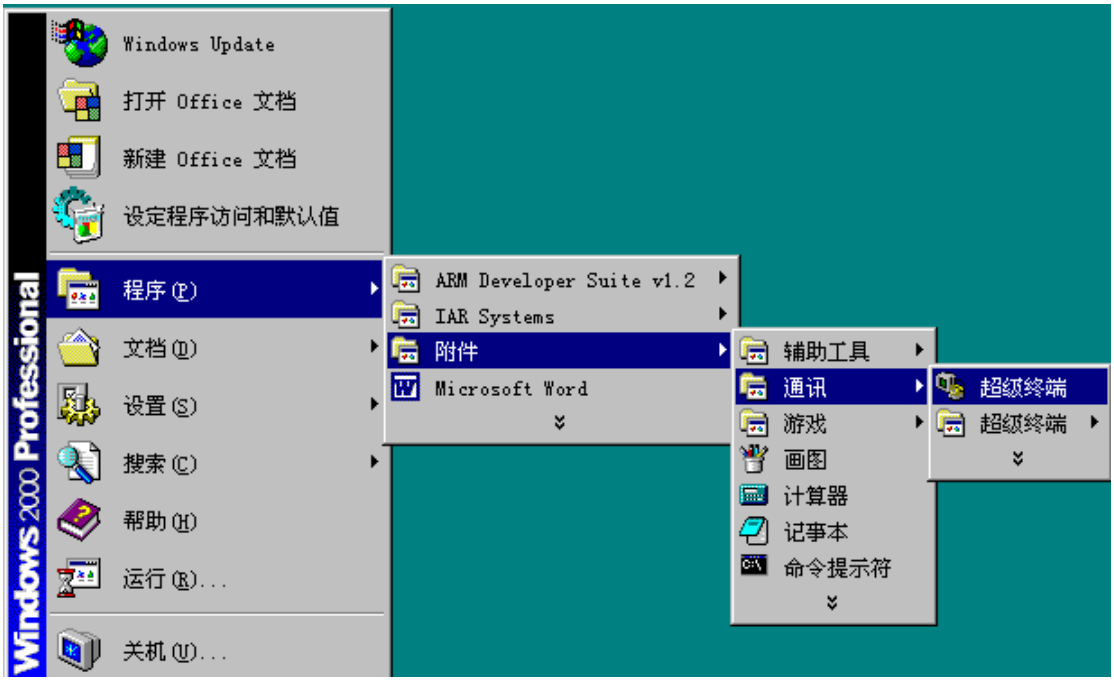
装载完成之后便可对该目录进行操作，在使用新的软盘及光盘前必须退出该目录，使用卸载命令进行卸载，方可使用新的软盘及光盘，否则系统不会承认该软盘的，光盘在卸载前是不能用光驱面板前的弹出键退出的。

以上仅为常用的Linux命令，关于Linux命令更详细的说明，请参考相关资料。

附录二 超级终端的使用和配置

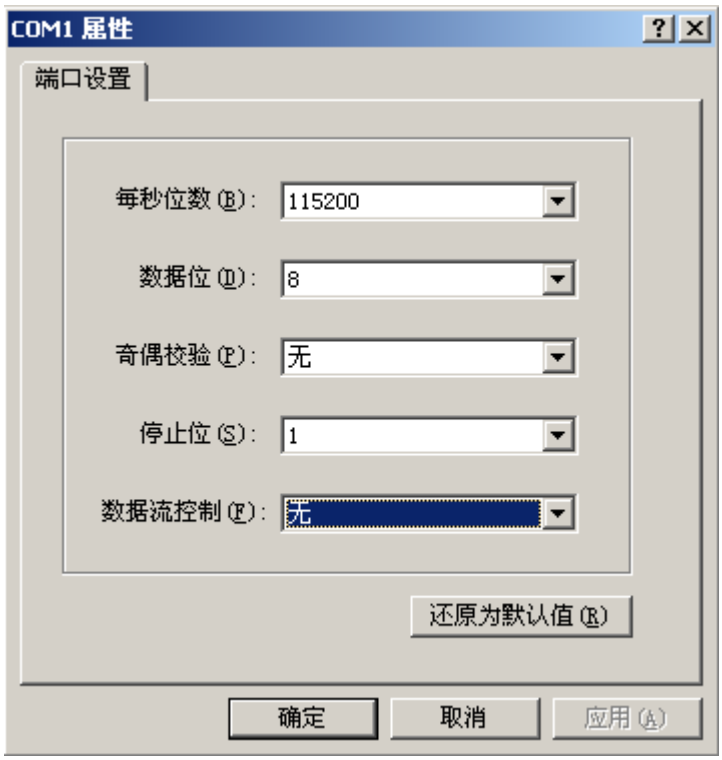
在Windows操作系统中有一个超级终端的应用程序，该程序可以完成对PC机的通信终端如串行口、网络接口的读/写操作，在本实验系统中，我们多次使用了超级终端，以下对其使用与配置方法作一个简单介绍。

启动超级终端，开始菜单->程序->附件->通信->超级终端

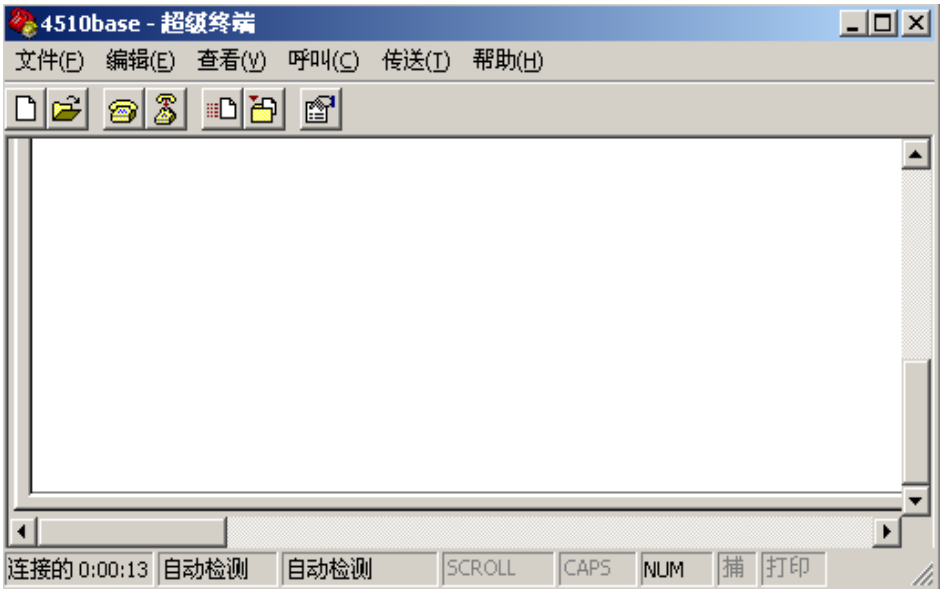


任意输入一个名称后点击确定按钮：

设置连接使用为串口1（COM1）(也可使用COM2，根据PC机的实际连接确定)，点击确定按钮进入下一步：



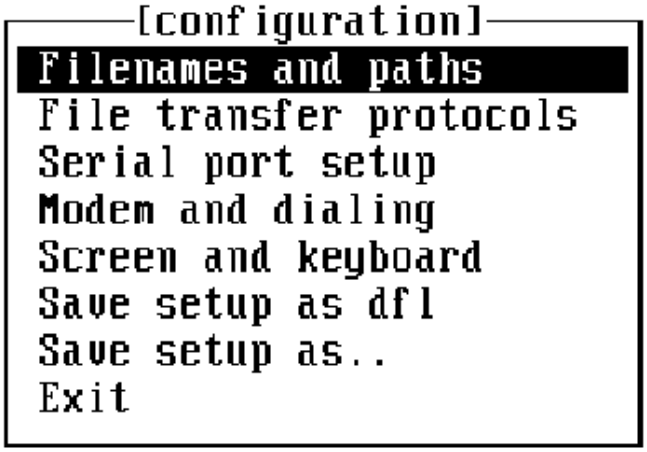
请按上图中的设定值进行设定：设置串口每秒位数 115200，数据位 8，奇偶校验 无，停止位 1，数据流控 无。点击确定按钮，如下图所示：



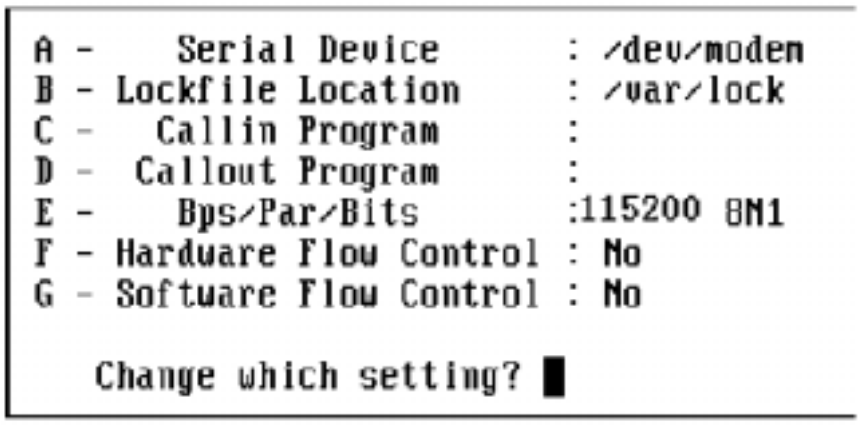
到此为止，超级终端的配置完毕，可以显示从串行口接收的信息，同时可以把用户从键盘输入的信息发送到串行口。

附录三 MINICOM 的使用和配置

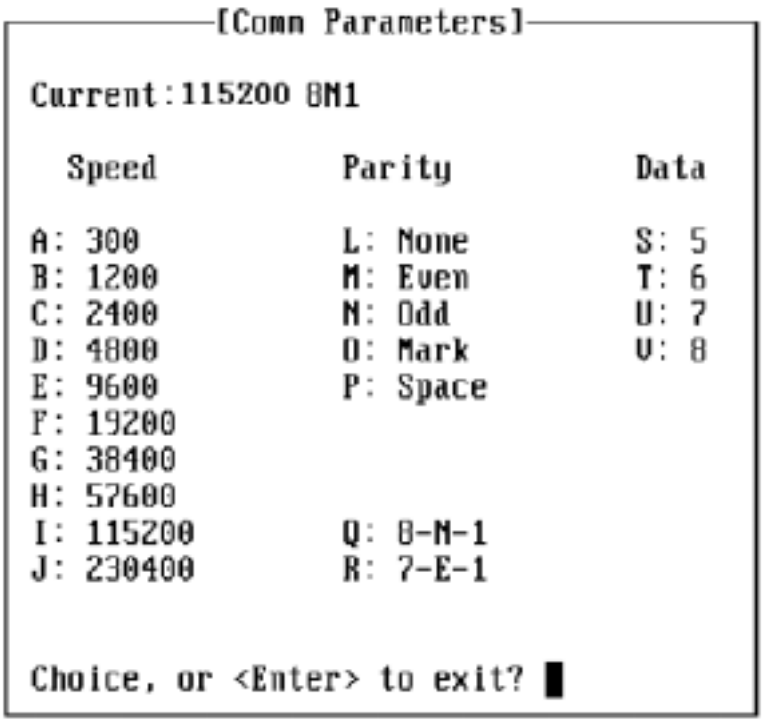
在Linux操作系统下，也有一个类似windows超级终端的工具，叫做minicom，minicom是安装REDHAT时安装的软件，以下对minicom的使用与配置作以简介。
在Linux操作系统提示符下键入：
$ minicom –s
显示如下对话框；



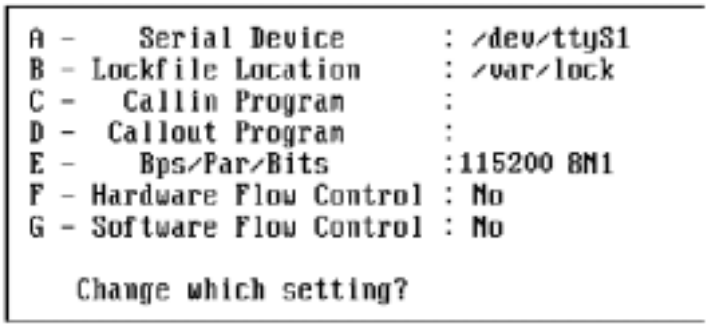
minicom 中最重要的操作就是对其进行配置的修改，选择第三项“Serial port setup”，则弹出如下对话框：



键入E 则弹出如下框，可改变波特率：



若要使用PC 机的串口2 来接板子的串口1 做监控，则要在串口配置框中选择A，即“ Serial Device ”，则原来的配置框第一行进入编辑模式，将原来的/dev/modem 改为如下的：/dev/ttyS1，即串口2。



退出配置框只需连续按ESC 键即可返回。

完成以上配置以后，在 Linux 的命令提示符下键入：

```
$ minicom
```

则打开 minicom 应用程序，使用方法与 Windows 下的超级终端相似。